Diseño de Circuitos Lógicos Combinatorios utilizando Programación Genética

Eduardo Serna Pérez

Maestría en Inteligencia Artificial
Universidad Veracruzana - LANIA

Asesora: Dra. Katya Rodríguez Vázquez
Departamento de Ingeniería de Sistemas Computacionales
y Automatización IIMAS-UNAM

Co-asesor: Dr. Carlos A. Coello Coello
Centro de Investigación y Estudios Avanzados del IPN

Revisor: Dr. Saúl Daniel Santillán Gutiérrez
Centro de Diseño y Manufactura, Facultad de Ingeniería UNAM

Enero, 2001

Resumen

El diseño de circuitos lógicos combinatorios ha sido una de las áreas abiertas a la investigación en los últimos años. Este interés es debido a lo complejo que resulta su diseño y especialmente su simplificación. Por años se ha utilizado el álgebra booleana como la manera más sencilla de elaborar e implementar el diseño de circuitos lógicos. Posteriormente se desarrollaron diversos métodos de inspección visual que fueron dirigidos principalmente a facilitar su simplificación como los mapas de Karnaugh y el método de Quine-McCluskey.

En años recientes distintas técnicas de búsqueda, optimización y aprendizaje, cuya inspiración radica en las teorías de la evolución y la selección natural, han sido aplicadas para resolver este tipo de problemas. A estas técnicas se les conoce como Computación Evolutiva.

Las técnicas evolutivas han comenzado a incursionar en el diseño de circuitos debido principalmente a su poder exploratorio, ya que permiten evaluar diversas regiones de un espacio de diseño y encontrar varias soluciones a problemas complejos con relativa eficiencia. Actualmente a esta área se le conoce como Hardware Evolutivo.

Diversas técnicas evolutivas han sido empleadas en hardware evolutivo. La implementación presentada en este trabajo esta situada dentro de la Programación Genética, la cual propone la utilización de cadenas prefijas como individuos para el diseño de circuitos lógicos combinatorios. La programación genética es un paradigma interesante y adecuado para este problema debido principalmente a su alto poder exploratorio.

En la implementación propuesta se realiza una serie de experimentos con distintos tipos de circuitos, los cuales varían en grado de complejidad y dimensionalidad. Dichos experimentos tienen como objetivo verificar la capacidad de elaboración de diseños lógicos factibles, que finalmente serán comparados con otras técnicas tradicionales y evolutivas. Además, el algoritmo propuesto muestra una gran capacidad para simplificar diseños lógicos en un tiempo bastante razonable y un uso de recursos computacionales bajo.

Agradecimientos

Agradezco a Dolores su apoyo y comprensión.

A Guadalupe, Fernando, Ximena y Fersillo les expreso un especial agradecimiento por su apoyo continuo y permitirme vivir en Coatepec y convivir en la ciudad de México.

Gracias a mis dos supervisores, Dra. Katya Rodríguez Vázquez y Dr. Carlos Coello Coello por su ayuda y orientación a lo largo de este trabajo de tesis.

También quiero darles las gracias a Narely y Nora, por siempre demostrarme su amistad incondicional.

Agradezco el apoyo otorgado por CONACyT a través de una beca para cursar la Maestría en Inteligencia Artificial del LANIA y la Universidad Veracruzana.

Gracias al Departamento de Ingeniería de Sistemas Computacionales y Automatización del IIMAS-UNAM, por permitirme realizar mi trabajo de tesis en sus instalaciones. Así como a todas aquellas personas que tuve el agrado de conocer en el IIMAS-UNAM.

Asimismo, agradezco la beca terminal otorgada por CONACyT para concluir esta tesis de maestría, a través del proyecto CONACyT No. J34900-A.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1 Motivación

El *diseño de circuitos lógicos combinatorios* ha sido uno de los problemas básicos de la electrónica. Desde sus inicios, la electrónica utilizó técnicas tradicionales basadas en el álgebra booleana para diseñar circuitos. Posteriormente conforme las demandas de procesamiento fueron creciendo, las necesidades de construir circuitos más eficientes y menos costosos, se volvió más importante. En este punto se comenzaron a desarrollar distintas técnicas que permitieran desarrollar circuitos más baratos y en menor tiempo. Los métodos visuales como los mapas de Karnaugh dieron paso a técnicas más fáciles de trasladarse a una computadora, como el método de Quine-McCluskey. Asimismo, la síntesis de circuitos pasó a volverse cada vez más importante, dando pie a sofisticadas técnicas heurísticas de búsqueda como las incorporadas en ESPRESSO.

Sin embargo, la complejidad del problema de diseño de circuitos combinatorios es tal que ha permanecido como una área abierta de investigación a través de los años.

Recientemente se ha comenzado a utilizar otro tipo de técnicas de búsqueda, optimización y aprendizaje, cuya inspiración ha sido obtenida de las ideas de la selección natural y las teorías de la evolución basadas en el paradigma Neo-Darwiniano. A estas técnicas se les conoce como *Computación Evolutiva.* La computación evolutiva realiza una simulación de los procesos evolutivos en las computadoras. Estas técnicas evolutivas han comenzado a ser muy populares, debido principalmente a su alto poder exploratorio y a su paralelismo implícito.

El problema de diseñar circuitos ha comenzado a ser explorado por las distintas técnicas evolutivas con resultados aceptables que han permitido explorar diversas regiones del espacio de diseño. A esta área se le conoce como *Hardware Evolutivo*, existiendo diversas formas de diseñar circuitos evolutivos, ya sea en línea (o sea, directamente en hardware reconfigurable), o fuera de línea (usando simulaciones). Sin embargo, las simulaciones resultan más económicas y nos permiten prevenir posibles conflictos que podrían resultar bastante costosos.

El hardware evolutivo puede ser dividido en dos principales grupos, la evolución intrínseca que es hardware reconfigurable, y la extrínseca que son las simulaciones. Cada una de estas categorías pueden ser subdivididas en sistemas análogos y digitales.

Este trabajo de tesis se encuentra situado en el ámbito extrínseco de diseño donde se pretende construir circuitos lógicos combinatorios a nivel de compuertas; es decir, utilizar sólo las compuertas lógicas AND, OR, NOT y XOR, y prescindiendo de la creación de módulos automáticos (nivel de funciones).

Además se pretende utilizar una representación distinta a la que generalmente se emplea en otras implementaciones evolutivas. Se propone una representación de individuos con cadenas de expresiones prefijas que permitan construir y evaluar circuitos lógicos utilizando la *Programación Genética* para realizar el proceso de evolución.

La técnica es probada con algunos ejemplos extraídos de la literatura de diseño de circuitos lógicos combinatorios. Los circuitos varían en complejidad, número de variables y número de salidas. Asimismo, los resultados extraídos son comparados con otras técnicas (evolutivas y clásicas) realizando paralelamente un análisis de los datos obtenidos.

1.2 Objetivo

El objetivo principal planteado en este trabajo de tesis se describe a continuación:

La aplicación de la representación prefija (Programación Genética) en el diseño de circuitos lógicos (de una y múltiples salidas), analizando los beneficios de la misma, la calidad de los diseños producidos y el poder exploratorio de las diversas regiones del espacio de diseño de circuitos.

1.3 Estructura de la tesis

El documento está organizado en 7 capítulos, los cuales se describen brevemente a continuación:

El *capítulo 1* presenta una introducción general al trabajo de tesis, la motivación que nos condujo a realizar la implementación propuesta y una breve descripción del contenido de cada uno de los capítulos de la tesis.

En el *capítulo 2* se presentan las teorías y conceptos principales que involucran el diseño de circuitos y las distintas técnicas que han sido desarrolladas para el diseño de circuitos lógicos combinatorios.

En el *capítulo 3* se da una breve introducción a la computación evolutiva, sus bases teóricas y los diversos paradigmas existentes, haciendo énfasis en la programación genética. Asimismo se describe brevemente lo que es el área de hardware evolutivo.

En el *capítulo 4* se explica el funcionamiento de la técnica PG Prefija, la forma en cómo trabaja la implementación, su representación y la técnica evolutiva utilizada. Además, se muestra un ejemplo del proceso de evolución generado para diseñar un circuito.

En el *capítulo 5* se muestran los diversos resultados obtenidos y el análisis de algunos ejemplos de circuitos de una salida que fueron diseñados con la técnica propuesta.

En el *capítulo 6* se describen las extensiones que se realizaron para resolver circuitos de múltiples salidas, así como los resultados obtenidos y el análisis de algunos ejemplos.

Finalizando con el *capítulo 7*, se presentan las conclusiones finales del trabajo de tesis realizado y las perspectivas futuras del mismo.

Capítulo 2

Circuitos lógicos

2.1 Implementación de un sistema lógico

Entre 1847 y 1854, el matemático George Boole [65] elaboró interesantes estudios en el campo de la lógica, la matemática y los procesos del pensamiento, desarrollando novedosas ideas en métodos lógicos. Confiado en el razonamiento simbólico que había derivado de sus investigaciones matemáticas, Boole argumentaba la importancia de aliar la lógica con las matemáticas. Boole remarcaba la analogía entre los símbolos algebraicos y aquellos que pueden representar formas lógicas y silogismos. Esto lo llevó a la creación del álgebra de variables lógicas, ahora conocida como álgebra booleana [7, 62, 36].

2.1.1 Algebra booleana

El álgebra booleana difiere del álgebra ordinaria en que las variables lógicas sólo pueden tomar dos posibles valores, 0 o bien 1. Las variables lógicas o booleanas se emplean para representar el nivel de voltaje existente en un alambre o en las terminales de entrada o salida de un circuito lógico [66, 49, 58]. Por ejemplo, en un sistema digital el valor booleano de 0 podría asignarse a un voltaje de 0.0 a $0.8V$, mientras que para el valor booleano de 1 el voltaje podría variar de 2.0 a $5.0V$.

De tal forma el 0 y el 1 booleanos no representan en este caso números, sino el estado de una variable lógica con respecto al voltaje, conocido como *nivel lógico* (Tabla 2.1).

Al igual que en el álgebra ordinaria, en el álgebra booleana se emplean símbolos alfabéticos a los cuales se les asocia un valor, que en este caso es el dominio binario $\{0,\ 1\}$, teniendo $A = 0$ o bien $A = 1$. De tal forma que

0	1
Falso / False	Verdadero/True
Bajo/Low	Alto/High
No	Si

Tabla 2.1: La lógica digital emplea más términos como sinónimos de 0 y 1.

una variable lógica obtiene su valor dependiendo de la relación funcional que existe con otras variables y operadores lógicos. En el álgebra booleana existen 3 operadores lógicos básicos.

Suponemos que $A = 1$ y $B = 0$ representan variables lógicas:

1. **Disyunción:** también llamada Adición Lógica o simplemente Operación OR. Su símbolo alternativo es $(+)$. El resultado será siempre 1 si existe alguna variable con 1. En la expresión: $A + B = 1$

2. **Conjunción:** también llamada Producto Lógico o simplemente Operador AND. Su símbolo alternativo es $(\cdot)$ pero por lo general se omite. El resultado será siempre 1 si todas las variables son 1. En la expresión: $A \cdot B = 0$ o bien $AB = 0$

3. **Negación:** también llamada Inversión Lógica, Complemento o simplemente Operador NOT. Su símbolo alternativo es $(')$. El resultado siempre invierte el nivel lógico, es decir de 0 a 1 y viceversa. En la expresión: $A' = 0$, $B' = 1$

El álgebra booleana se utiliza para expresar los efectos que los diversos circuitos lógicos o digitales ejercen sobre las entradas lógicas, que asumen la salida como un significado lógico. También se usa para manipular variables lógicas con el objeto de determinar el mejor método de ejecución de cierta función en un circuito. Los circuitos que realizan las operaciones lógicas básicas se denominan compuertas lógicas.

2.1.2 Compuertas lógicas

Las compuertas lógicas son bloques de hardware que producen señales de salida 1 ó 0 binarias cuando los requerimientos lógicos de entrada son satisfechos. En los sistemas de cómputo digitales se utiliza por lo general una variedad de compuertas lógicas también llamadas circuitos lógicos. Cada compuerta tiene un símbolo gráfico distinto y su funcionamiento puede describirse por medio de una expresión algebraica. Además la relación de entrada/salida de las variables binarias para cada compuerta puede representarse en forma tabular por medio de una tabla de verdad.

A	B	Z
0	0	?
0	1	?
1	0	?
1	1	?

Tabla 2.2: Tabla de verdad de 2 entradas / 1 salida.

Una tabla de verdad muestra la relación binaria que existe entre las diversas combinaciones de los niveles lógicos de las N variables de entrada y las M variables de salida. El número de combinaciones de entrada será igual a 2^N para una tabla de verdad de N entradas. En cada tabla de verdad las combinaciones posibles de niveles lógicos 0 y 1 para las entradas (A, B) se enlistan del lado izquierdo y el nivel lógico de salida se enlista del lado derecho como lo muestra la Tabla 2.2.

La compuerta OR

Es un circuito lógico que tiene 2 o más entradas y cuya salida es igual a la adición lógica de las entradas. La Figura 2.1 muestra la tabla de verdad y el símbolo correspondiente a una compuerta OR de dos entradas.

Las entradas A y B son niveles de voltaje lógicos y la salida X es un nivel de voltaje lógico cuyo valor es el resultado de la operación OR de A y B; esto es, $X = A + B$. En otras palabras, la compuerta OR opera de tal forma que su salida sea ALTA (nivel lógico 1) si cualquiera de las entradas A, B o ambas están en un nivel lógico 1. De lo contrario será BAJA (nivel lógico 0).

A	B	$X = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

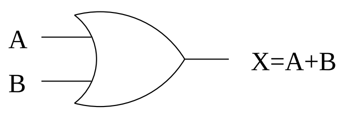

Figura 2.1: Tabla de verdad y compuerta OR de dos entradas.

La compuerta AND

Es un circuito lógico cuya salida es igual al producto de las 2 o más entradas lógicas; es decir, $X = AB$. En otras palabras, la compuerta AND opera en forma tal que su salida sea ALTA sólo cuando todas sus entradas están en un nivel lógico 1. En todos los demás casos la salida de la compuerta AND será BAJA. La Figura 2.2 muestra la tabla de verdad y el símbolo correspondiente a una compuerta AND de dos entradas.

A	B	$X = AB$
0	0	0
0	1	0
1	0	0
1	1	1

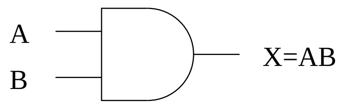

Figura 2.2: Tabla de verdad y compuerta AND de dos entradas.

La compuerta NOT

Este circuito siempre tiene sólo una entrada y su nivel lógico de salida siempre es contrario al nivel lógico de esta entrada. La Figura 2.3 muestra la tabla de verdad y el símbolo correspondiente a una compuerta NOT, la cual se le conoce más comúnmente como INVERSOR.

A	$X = A'$
0	1
1	0

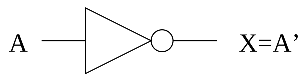

Figura 2.3: Tabla de verdad y compuerta NOT.

Cualquier circuito lógico, sin importar qué tan complejo sea, puede describirse completamente mediante el uso de las operaciones que se definieron anteriormente, ya que las compuertas OR, AND y NOT son los elementos básicos de todos los sistemas digitales.

2.1.3 Circuito OR-exclusivo

Un circuito lógico especial que aparece con frecuencia en los sistemas digitales es el circuito OR-exclusivo conocido comúnmente como XOR. La expresión de salida de este circuito es $X = A'B + AB'$. En la Figura 2.4, la tabla de verdad muestra que $X = 1$ en dos casos: $A = 0$, $B = 1$ y $A = 1$,

A	B	$X = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

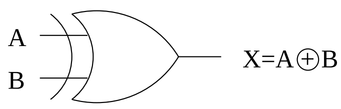

Figura 2.4: Tabla de verdad y compuerta XOR de dos entradas.

$B = 0$. Es decir, este circuito produce una salida ALTA siempre que las dos entradas están en niveles opuestos.

Esta combinación específica de compuertas lógicas ocurre con mucha frecuencia y es de mucha utilidad en ciertas aplicaciones. De hecho, al circuito XOR se le ha dado un símbolo propio (Figura 2.4) y se le considera como otro tipo de compuerta lógica.

Una manera abreviada que se utiliza algunas veces para indicar la salida XOR es $X = A \oplus B$, donde el símbolo $\oplus$ representa la operación de la compuerta XOR:

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

Una compuerta XOR tienen 2 entradas, aunque pueden existir de más entradas, si bien éstas no son comunes, ya que la compuerta XOR es una de las más complicadas de construir en hardware y no es fácil modificar una compuerta XOR de dos entradas para añadirle entradas adicionales.

2.1.4 Teoremas booleanos

Hemos observado cómo se puede utilizar el álgebra booleana como auxiliar en el análisis de un circuito y expresar su operación matemáticamente.

A continuación estudiaremos los diversos teoremas booleanos (reglas) que nos pueden servir para simplificar las expresiones y los circuitos lógicos. Cualquier función lógica puede expresarse mediante las operaciones OR, AND y NOT. Los primeros ocho teoremas (Tabla 2.3) muestran la relación básica entre una sola variable y sí misma, o en conjunto con las constantes binarias.

Los teoremas que se presentan a continuación implican más de una variable y son similares a los del álgebra ordinaria. En cada teorema:

W, X, Y, Z son variables lógicas.

1) $X \cdot 0 = 0$	2) $X \cdot 1 = X$
3) $X \cdot X = X$	4) $X \cdot X' = 0$
5) $X + 0 = X$	6) $X + 1 = 1$
7) $X + X = X$	8) $X + X' = 1$

Tabla 2.3: Teoremas 1 al 8. Relación entre una sola variable.

Los teoremas 9 y 10 se denominan leyes conmutativas. Estas leyes indican que el orden en el cual operamos dos variables con OR y AND es intrascendente; el resultado es el mismo.

$$9)\ X + Y = Y + X \qquad 10)\ XY = YX$$

Los teoremas 11 y 12 son las leyes asociativas, las cuales afirman que se pueden agrupar las variables en una expresión AND o en una OR en la forma que se desee.

$$11)\ X + (Y + Z) = (X + Y) + Z = X + Y + Z$$
$$12)\ X(YZ) = (XY)Z = XYZ$$

El teorema 13 es la ley distributiva, la cual afirma que una expresión puede aplicarse multiplicando término a término, como en el álgebra ordinaria. Además podemos factorizar una expresión; es decir, si se tiene una suma de dos o más términos cada uno de los cuales contiene una variable común, ésta se puede factorizar.

$$13a)\ X(Y + Z) = XY + XZ$$
$$13b)\ (W + X)(Y + Z) = WY + XY + WZ + XZ$$

Los teoremas 14, 15, 16 y 17 no tienen equivalentes en el álgebra ordinaria, pero son muy útiles en la manipulación de expresiones booleanas.

$$14)\ X + XY = X \qquad 15)\ X + X'Y = X + Y$$
$$16)\ X + YZ = (X + Y)(X + Z) \qquad 17)\ X'' = X$$

Teoremas de DeMorgan

Dos de los teoremas más importantes del álgebra booleana fueron enunciados por el eminente matemático Augusto DeMorgan. Los teoremas de DeMorgan son de extrema utilidad en la simplificación de expresiones en las cuales se invierte un producto o suma de variables.

$$18)\ (X+Y)' = X'Y' \quad 19)\ (XY)' = X' + Y'$$

El teorema 18 afirma que cuando se invierte la suma OR de dos variables, esta inversión es la misma que la de cada variable en forma individual y luego la operación con AND de estas variables invertidas. El teorema 19 expresa que, cuando se invierte el producto AND de dos variables, esto equivale a invertir cada variable en forma individual y luego operarlas con OR.

Aunque estos teoremas se han enunciado en términos de variables sencillas X y Y, son igualmente válidos en situaciones donde X y/o Y son expresiones que contienen más de una variable. Por ejemplo, apliquémoslos en la expresión $(AB' + C)'$ obteniendo $(AB')'C'$.

2.2 Diseño de circuitos lógicos combinatorios

Un circuito combinatorio [49, 66] es un arreglo de compuertas lógicas con un conjunto de entradas y salidas. En cualquier momento dado, los valores binarios de las salidas son una combinación binaria de las entradas. En la Figura 2.5, se muestra un diagrama de bloque de un circuito combinatorio. Las N variables de entrada vienen de una fuente externa: las M variables de salida van a un destino externo que representan el nivel de salida deseado para un circuito lógico y entre éstas hay una interconexión de compuertas lógicas.

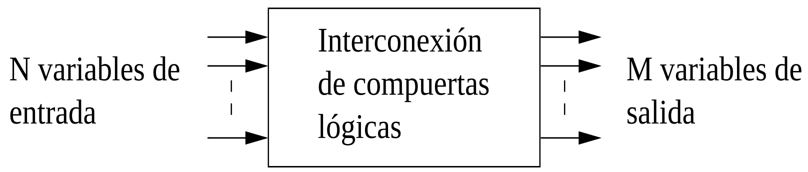

Figura 2.5: Diagrama de bloque.

El diseño de circuitos combinatorios parte del planteamiento verbal del problema y termina con un diagrama de circuito lógico. El procedimiento comprende los siguientes pasos:

1. Se establece el problema.
2. Se asignan símbolos literales a las variables de entrada y salida.

3. Se deriva la tabla de verdad que define la relación entre entradas y salidas.

4. Se obtienen las funciones booleanas simplificadas para cada salida.

5. Se traza el diagrama lógico.

2.2.1 Formas estándar para funciones lógicas

Una función lógica [65, 58] puede derivarse de la tabla de verdad en la forma de suma de productos también llamada suma de minitérminos o por un producto de sumas conocido como producto de maxitérminos. Las relaciones entre estos modos de representación se expresan a continuación para una función seleccionada arbitrariamente.

1. Suma de productos.

$$F = A'B'C' + A'BC' + A'BC + ABC' + ABC$$
$$F(A,B,C) = \Sigma m(0,2,3,6,7)$$

2. Producto de sumas.

$$F = (A + B + C')(A' + B + C)(A' + B + C')$$
$$F(A,B,C) = \Pi M(1,4,5)$$

En la tabla de verdad (Tabla 2.4), cada fila tiene asignado el número que resultaría si las entradas lógicas bajo A, B y C fuesen dígitos binarios.

Fila	A	B	C	F(A,B,C)
0	0	0	0	1
1	0	0	1	0
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

Tabla 2.4: Tabla de verdad de la función seleccionada

Suma de minitérminos

Señalamos que la Tabla 2.4 indica en la fila 0 que $F = 1$ cuando $A = 0$, $B = 0$ y $C = 0$. Si escribimos la función como una suma de minitérminos, es claro asegurar que para $F = 1$ con $A = B = C = 0$ implicaría incluir el minitérmino $A'B'C'$. La inclusión de dicho minitérmino nos permite asegurar que $F = 1$ cuando $A = B = C = 0$, sin importar los términos que se añadan posteriormente, ya que " $1 + cualquier\ cosa = 1$ ". En conjunto, una función expresada por suma de minitérminos incluye precisamente a aquellos cuyos números coinciden con las filas de la tabla de verdad para los que $F = 1$.

Producto de maxitérminos

Los maxitérminos son precisamente los de las filas para los cuales $F = 0$. Consideremos a título de ejemplo la fila 1 en la Tabla 2.4. Para asegurarnos que $F = 0$ cuando $A = 0$, $B = 0$ y $C = 1$, habrá de incluir el maxitérmino $(A + B + C')$. Por contener este maxitérmino, aseguraremos que $F = 0$ para $A = 0$, $B = 0$ y $C = 1$, sin importar que más tarde se añadan otros maxitérminos, ya que " $0 \cdot cualquier\ cosa = 0$ ".

Supongamos que queremos expresar como producto de maxitérminos una función representada por suma de minitérminos. Necesitamos anotar solamente aquellos maxitérminos cuyos números no aparezcan en la lista de minitérminos. Por ejemplo, supongamos la función de tres variables $F = \Sigma m(0, 3, 6, 7)$. La misma función puede representarse también como $F = \Pi M(1, 2, 4, 5)$.

2.2.2 El proceso de diseño

Una vez establecidas las funciones lógicas, demostraremos el proceso de diseño [49] de los circuitos lógicos combinatorios, tomando en cuenta los pasos previamente mencionados. El éxito del proceso de diseño radica en la experiencia y familiaridad del diseñador con los circuitos digitales. La capacidad para correlacionar una tabla de verdad o un conjunto de funciones booleanas con una tarea de procesamiento de información es casi un arte que se adquiere con la experiencia. A continuación se presenta un ejemplo de circuito aritmético simple. Este circuito sirve como una unidad básica de construcción para circuitos aritméticos más complicados.

Sumador completo

El sumador completo es un circuito combinatorio que realiza la suma aritmética de tres bits de entrada. Consiste de tres entradas y dos salidas. Dos de las variables de entrada, denotadas con x y y, representan los dos bits significativos a sumarse. La tercera entrada, z, representa el acarreo de la posición menos significativa previa. Las dos salidas son necesarias porque la suma aritmética de tres dígitos binarios fluctúa entre 0 y 3 y el 2 o el 3 binario, necesitan dos dígitos. Las dos salidas se designan por los símbolos S (por suma) y C (por acarreo). La variable binaria S da el valor del bit menos significativo de la suma. La variable binaria C da el acarreo de la salida.

Entradas			Salidas	
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Tabla 2.5: Tabla de verdad para el sumador completo.

La tabla de verdad para el sumador completo se muestra en la Tabla 2.5. Los ocho renglones bajo las variables de entrada designan todas las posibles combinaciones que pueden tener las variables binarias. El valor de las variables de salida se determina de la suma aritmética de los bits de entrada. Cuando todos los bits de entrada son 0 la salida S es 0, la salida S es igual a 1 cuando sólo una entrada es igual a 1 o cuando las tres entradas son iguales a 1. La salida C tiene un acarreo de 1 si dos o tres entradas son iguales a 1.

Basándose en la tabla de verdad se derivan las funciones lógicas utilizando la suma de minitérminos y se simplifican las expresiones resultantes usando álgebra booleana.

S	$= x'y'z + x'yz' + xy'z' + xyz$	teorema 11 y 13a
	$= x'(y'z + yz') + x(y'z' + yz)$	sustituir términos con compuertas xor y xnor
	$= x'(y \oplus z) + x(y \oplus z)'$	y reemplazamos términos con compuerta xor
	$= x \oplus y \oplus z$	

C	$= x'yz + xy'z + xyz' + xyz$	teorema 11 y 13a
	$= z(x'y + xy') + xy(z' + z)$	sustituir términos con compuerta xor y teorema 8
	$= z(x \oplus y) + xy$	

El diagrama lógico del sumador completo se muestra en la Figura 2.6. Existe una particularidad muy interesante: la compuerta $x \oplus y$ se repite en ambas salidas y al momento de realizar el diagrama esta salida puede ser reutilizada como entrada para las compuertas que lo necesiten por medio de una conexión extra. Esta clase de reutilización de compuertas es una característica muy deseada por los diseñadores de circuitos, ya que permiten un ahorro al momento de producirlas. Desafortunadamente, no resulta tan obvio encontrar dichas similitudes.

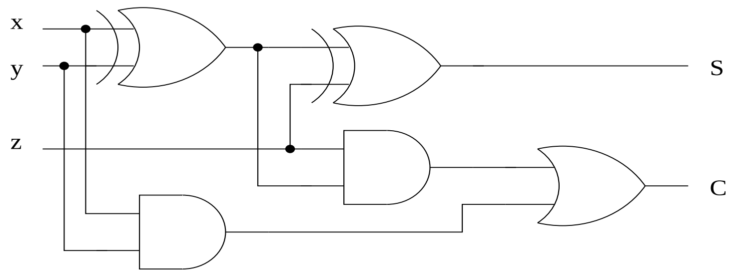

Figura 2.6: Diagrama lógico del sumador completo.

2.3 Simplificación de circuitos lógicos

Una vez que la expresión para un circuito lógico se haya obtenido, podemos reducirla a una forma más simple que contenga menos términos o variables en uno o más términos. La nueva expresión puede utilizarse entonces para activar un circuito que sea equivalente al original pero que contenga menos compuertas y conexiones. Ya que ambos circuitos ejecutan la misma lógica, debe ser obvio que el circuito más simple sea más viable debido a que contiene menos compuertas y, por tanto, será de menor tamaño y menor costo que el original. Además, la confiabilidad del circuito mejorará ya que hay menos interconexiones que pueden constituirse en fallas potenciales del circuito.

La función lógica puede aparecer en muchas formas diferentes cuando se expresa algebraicamente y puede simplificarse por medio de las relaciones básicas del álgebra booleana. Sin embargo, ese procedimiento es algunas veces difícil de deducir y requiere una gran experiencia por parte del diseñador. Esta limitante ha traído como consecuencia el desarrollo de métodos que permitan una simplificación eficiente y eficaz al momento de diseñar circuitos. Los métodos como los Mapas de Karnaugh [41] y el de Quine-McCluskey [56, 50], cubren parcialmente estas limitantes.

2.3.1 Simplificación algebraica

Los teoremas del álgebra booleana se pueden utilizar para simplificar expresiones lógicas. Desafortunadamente, no siempre es obvio qué teorema aplicar a fin de producir el resultado más simple. Además, no existe manera de indicar si la expresión simplificada está en su forma más simple. Así, la simplificación algebraica [34, 70] con frecuencia se convierte en un proceso de ensayo y error. Sin embargo, con experiencia uno puede llegar a obtener resultados razonablemente buenos.

El siguiente ejemplo ilustrará muchas de las maneras en que pueden aplicarse los teoremas booleanos al intentar simplificar una expresión. La simplificación algebraica consta de 3 etapas:

1. La expresión se desarrolla en forma de suma de productos o producto de sumas.

2. Los términos de la expresión se verifican para ver si hay factores comunes y se realiza la factorización siempre que sea posible. Con suerte, la factorización da como resultado la eliminación de uno o más términos.

3. Se recurre al resto de los teoremas para buscar eliminar factores.

Tenemos la siguiente expresión de salida

$$z = abc + ab'(a'c')'$$

Por lo general conviene eliminar todos los signos inversores de mayor tamaño por medio de los teoremas de DeMorgan y luego multiplicamos todos los términos.

$$\begin{aligned} z &= abc + ab'(a'' + c'') && \text{teorema 19 leyes de DeMorgan} \\ &= abc + ab'(a + c) && \text{teorema 17 inversores dobles} \\ &= abc + ab'a + ab'c && \text{teorema 13 leyes distributivas} \\ &= abc + ab' + ab'c && \text{teorema 3 } aa = a \end{aligned}$$

Con la expresión en la forma de suma de productos, debemos buscar variables comunes entre los diversos términos con fines de su factorización. El primero y tercer términos anteriores tienen ac en común, que puede factorizarse como sigue:

$$z = ac(b + b') + ab'$$

Ya que $b + b' = 1$, entonces

$$\begin{aligned} z &= ac(1) + ab' \\ &= ac + ab' \end{aligned}$$

Ahora podemos factorizar a, lo cual produce

$$z = a(c + b')$$

Este resultado ya no se puede simplificar más. Esta expresión es mucho más simple que el original. Con frecuencia se encuentran funciones lógicas susceptibles de simplificarse, aplicando los teoremas booleanos, aunque nunca se tenga la certeza de haber llegado al resultado más simple.

2.3.2 Método del mapa de Karnaugh

El mapa de Karnaugh es un dispositivo gráfico que se utiliza para simplificar una ecuación lógica mediante un proceso simple y ordenado. Aunque un mapa de Karnaugh, o mapa K, se puede utilizar para resolver problemas con cualquier número de variables de entrada, su utilidad práctica se limita a seis variables.

El mapa es un conjunto de 2^n cuadrículas, que representan las posibles combinaciones de los valores de las n variables binarias. Como las cuadrículas corresponden a combinaciones que se van a usar para escribir información, las combinaciones se escriben habitualmente en la parte superior de las cuadrículas.

La tabla de verdad (Tabla 2.6) da el valor de la salida x para cada combinación de valores de entrada. El mapa K proporciona la misma información en diferente formato. Cada caso o minitérmino en la tabla de verdad corresponde a un cuadrado en el mapa de Karnaugh (Figura 2.7). Los cuadrados del mapa K se marcan de modo que los cuadrados horizontales adyacentes difieren únicamente en una variable. Análogamente, los cuadrados verticales adyacentes difieren sólo en una variable.

Fila	a	b	c	x		
0	0	0	0	1	$\rightarrow$	$a'b'c'$
1	0	0	1	1	$\rightarrow$	$a'b'c$
2	0	1	0	1	$\rightarrow$	$a'bc'$
3	0	1	1	0		
4	1	0	0	0		
5	1	0	1	1	$\rightarrow$	$ab'c$
6	1	1	0	0		
7	1	1	1	0		

Tabla 2.6: Tabla de verdad de un circuito $F(a, b, c) = \Sigma m(0, 1, 2, 5)$.

La expresión de la salida x se puede simplificar adecuadamente combinando los cuadros en el mapa K que contengan unos. El proceso para combinar estos unos se denomina solapamiento y consiste en agrupar las casillas en 2^n casillas adyacentes. Si dos casillas adyacentes contienen un 1, entonces, los correspondientes minitérminos difieren sólo en una variable. En tal caso, los dos términos se pueden fundir en uno eliminando esta variable (Figura 2.8). Entonces, por ejemplo los minitérminos $a'b'c$ y $ab'c$ se combinan para producir $b'c$. Este proceso se puede ampliar de varias formas.

	b'c'	b'c	bc	bc'
a'	1	1	0	1
a	0	1	0	0

Figura 2.7: $F(a, b, c) = \Sigma m(0, 1, 2, 5)$

	b'c'	b'c	bc	bc'
a'	1	1	0	1
a	0	1	0	0

Figura 2.8: $x = b'c + a'c'$

Primero, el concepto de adyacencia se puede extender para incluir el recubrimiento alrededor del borde del mapa. Por tanto, la casilla más alta de una columna es adyacente a la más baja, y la casilla más a la izquierda de la fila es adyacente a la que está más a la derecha.

Para intentar simplificar, primero hay que mirar el grupo mayor posible de minitérminos adyacentes. Si alguna casilla con 1 queda sin círculo, entonces hay que mirar sucesivamente grupos más pequeños. Cuando se señalan grupos con un círculo, se está permitiendo el uso del mismo 1 más de una vez (Figura 2.9).

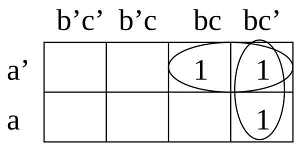

Figura 2.9: $f = a'b + bc'$

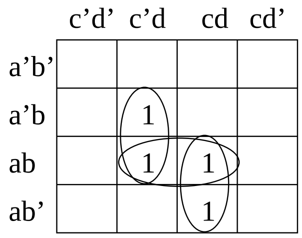

Figura 2.10: $f = bc'd + acd$

Si queda algún 1 aislado después de agrupar, entonces, cada uno de ellos se rodea con un círculo como si fuera un grupo de unos y es agregado tal cual a la expresión final. Por último, antes de pasar el mapa a una expresión simplificada, cualquier grupo de unos que esté completamente solapado por otros grupos se puede eliminar (Figura 2.10).

En las Figuras 2.11 y 2.12 se muestran dos ejemplos de simplificación de mapas de Karnaugh con cuatro variables de entrada.

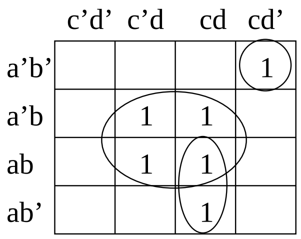

Figura 2.11: $f = a'b'cd' + acd + bd$

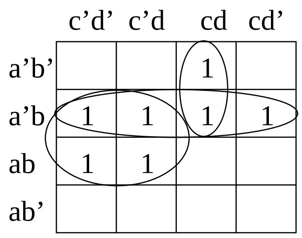

Figura 2.12: $f = a'b + bc' + a'cd$

En la Figura 2.11 se aprecia cómo una casilla queda aislada del resto de los grupos de unos y el grupo de cuatro logra eliminar dos variables. En la Figura 2.12, se cuenta con 2 formas en las que se pueden agrupar las casillas de cuatro y se muestra la forma como éstas comparten algunas casillas.

Los mapas de Karnaugh resultan muy útiles y sencillos para simplificar expresiones pequeñas (hasta con 4 variables), sin embargo nunca se asegura obtener una función óptima.

2.3.3 Método de Quine-McCluskey

Para más de cuatro variables, el método del mapa de Karnaugh se va haciendo cada vez más incómodo, Con cinco variables, se necesitan dos mapas de 4x4, con un mapa situado encima del otro, en tres dimensiones, para conseguir la adyacencia. ¡Seis variables requieren cuatro tablas de 4x4 en tres dimensiones! Una aproximación alternativa es una técnica tabular, denominada método de Quine-McCluskey.

Este método se explica mejor mediante un ejemplo. Consideremos la siguiente expresión:

$$F = abcd + abc'd + abc'd' + ab'cd + a'bcd + a'bcd' + a'bc'd + a'b'c'd$$

El primer paso es construir una tabla en la que cada fila corresponda a un término producto de la expresión. Los términos se agrupan de acuerdo con el número de variables complementadas. Es decir, empezamos con el término que tiene las n variables en complemento, si éste existe. Luego todos los términos con $n-1$ variables en complemento, y así sucesivamente hasta llegar al grupo de términos sin complementar.

Minitérmino	Indice	a	b	c	d	Combinar
$a'b'c'd$	1	0	0	0	1	$\surd$
$a'bc'd$	5	0	1	0	1	$\surd$
$a'bcd'$	6	0	1	1	0	$\surd$
$abc'd'$	12	1	1	0	0	$\surd$
$a'bcd$	7	0	1	1	1	$\surd$
$ab'cd$	11	1	0	1	1	$\surd$
$abc'd$	13	1	1	0	1	$\surd$
$abcd$	15	1	1	1	1	$\surd$

Tabla 2.7: Tabla para reducción de términos.

La Tabla 2.7 muestra la lista para nuestra expresión ejemplo, indicando con las líneas horizontales las agrupaciones. Para más claridad, cada término se representa con un 1 para cada variable sin complementar y con un 0 para cada variable complementada. Por tanto, agrupamos términos de acuerdo con el número de unos que contiene. La columna "índice" es simplemente el equivalente decimal y es útil para lo que se explicará más adelante.

El siguiente paso es encontrar todas las parejas de términos que difieren en sólo una variable. Es decir, todos los pares de términos que son iguales excepto que, una variable es 0 en uno de los términos y 1 en el otro. Debido a la manera en la que se agrupan términos, podemos hacerlo empezando con el primer grupo y comparar cada término del primer grupo con cada término del segundo grupo. Después comparamos cada término del segundo grupo con todos los términos del tercer grupo, y así sucesivamente.

Cuando se encuentra un emparejamiento, se coloca una marca en cada término, se combina el par eliminando la variable en la que difieren los dos términos, y se añade a la nueva lista. Entonces, por ejemplo, los términos $a'bcd'$ y $a'bcd$ se combinan para producir el término $a'bc$. Este proceso continúa hasta que se haya examinado la tabla original entera.

El resultado es una nueva tabla (Tabla 2.8) con los siguientes elementos:

Minitérmino	Indice	a	b	c	d	Combinar
$a'c'd$	1, 5	0	-	0	1	
abc'	12, 13	1	1	0	-	
$bc'd$	5, 13	-	1	0	1	$\surd$
$a'bc$	6, 7	0	1	1	-	
$a'bd$	5, 7	0	1	-	1	$\surd$
abd	13, 15	1	1	-	1	$\surd$
acd	11, 15	1	-	1	1	
bcd	7, 5	-	1	1	1	$\surd$

Tabla 2.8: Tabla para reducción de términos.

La nueva tabla se organiza en grupos, como indicamos antes, de la misma forma que la primera tabla. La segunda tabla se procesa, entonces, de la misma manera que la primera. Es decir se comprueban términos que difieren sólo en una variable y se produce un nuevo término para una tercera tabla. En este ejemplo, la tercera tabla (Tabla 2.9) contiene solamente un término bd.

En general, el proceso continuaría a través de sucesivas tablas hasta una en la que no haya emparejamientos. En este caso, hay implicadas tres tablas.

Una vez que se haya completado el proceso descrito anteriormente, tenemos que eliminar muchos de los posibles términos de la expresión. Aquellos términos que no hayan sido eliminados se usan para construir una matriz, como se ilustra en la Figura 2.13.

Minitérmino	Indice	a	b	c	d	Combinar
$a'c'd$	1, 5	0	-	0	1	
abc'	12, 13	1	1	0	-	
bd	5,13 y 7, 5	-	1	-	1	
$a'bc$	6, 7	0	1	1	-	
bd	5, 7 y 13, 15	-	1	-	1	
acd	11, 15	1	-	1	1	

Tabla 2.9: Tabla para reducción de términos.

Cada fila de la matriz corresponde a uno de los términos que no se han eliminado (no tiene marca) en la tabla usada anteriormente. Cada columna se corresponde con uno de los términos de la expresión original. Se coloca una X en cada intersección de una fila y una columna tal que el elemento de la fila sea " compatible " con el elemento de la columna.

	abcd	abc'd	abc'd'	ab'cd	a'bcd	a'bcd'	a'bc'd	a'b'c'd
bd	X	X			X		X	
a'c'd							$\boxed{X}$	X
a'bc					$\boxed{X}$	X		
abc'		$\boxed{X}$	X					
acd	$\boxed{X}$			X				

Figura 2.13: Tabla de solución para el problema con el método de Quine-McCluskey.

Es decir, las variables presentes en el elemento de la fila tienen el mismo valor que las variables presentes en el elemento de la columna. Después, se rodea con un círculo cada X que esté sola en una columna. Entonces, se sitúa un cuadrado alrededor de cada X en cualquier fila que tenga un círculo en la X. Si cada columna tiene ahora una X encerrada en un cuadrado o en un círculo, entonces ya se ha concluido, y los elementos de esa fila cuyas X estén marcadas constituyen la expresión mínima. En nuestro ejemplo:

$$F = a'c'd + a'bc + abc' + acd$$

En los casos en los que algunas columnas tengan un círculo pero no un cuadrado, se necesita un proceso adicional. Esencialmente, seguimos añadiendo elementos a la fila hasta que cubran todas las columnas.

Tras la eliminación de las variables, nos queda una expresión que es claramente equivalente a la expresión original. Sin embargo, puede haber términos redundantes en la expresión, como encontramos agrupaciones redundantes en los mapas de Karnaugh.

Capítulo 3

Computación evolutiva

La computación evolutiva es el término que se utiliza para englobar a todas aquellas técnicas de optimización, búsqueda y aprendizaje de máquina inspiradas en las teorías genéticas y de la evolución que gobiernan la adaptación de las especies en el planeta.

En este capítulo se proporcionan algunos antecedentes históricos de la genética y como éstos conducen al surgimiento del Neo-Darwinismo, el origen de la computación evolutiva y sus diversos paradigmas. Finaliza con una breve descripción del área de hardware evolutivo.

3.1 Antecedentes históricos de la genética

El Neo-Darwinismo es el conjunto de teorías que explican el origen de las especies en el planeta y cómo sus procesos de reproducción, mutación, competencia y selección permiten la adaptación y evolución de las especies.

Las principales teorías que engloban al Neo-Darwinismo son: el seleccionismo de Weismann, el origen de las especies de Darwin y la genética de Mendel.

3.1.1 Seleccionismo de Weismann

El zoólogo francés, Jean Baptiste Antoine de Monet (Caballero de Lamarck) comienza a publicar en 1801 su teoría para explicar la evolución [38]. 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.

Las teorías de Lamarck fueron rechazadas por el científico alemán Augusto Weismann a finales del siglo XIX, exponiendo su teoría del germoplasma [68]. Weismann argumentaba que una vez iniciada la fertilización, existen dos procesos de división celular. Uno dirigido a las células somáticas (somatoplasma) que desarrollan al cuerpo, y otro dirigido a las células germinales (germoplasma) que transmiten información hereditaria y que son el punto de inicio de la siguiente generación [4]. Concluyó que la selección natural era el único mecanismo capaz de cambiar al germoplasma[1], mientras que tanto el germoplasma como el ambiente pueden influenciar al somatoplasma[2].

3.1.2 Origen de las especies de Darwin

Una de las aportaciones más importantes en la teoría de la evolución fue hecha por el biólogo inglés Charles Darwin. En su libro *El Origen de las Especies* [23] (1859) explicó que una especie que no sufriera cambios se volvería incompatible con su ambiente, ya que éste tiende a cambiar con el tiempo. Asimismo, las similitudes entre hijos y padres observadas en la naturaleza, le sugirieron que ciertas características de las especies eran hereditarias y de generación en generación ocurrían cambios cuya principal motivación era hacer a los nuevos individuos más aptos para sobrevivir.

Darwin creía que la evolución era un largo proceso de adaptación al medio ambiente donde existía una competencia por la supervivencia, existiendo especies con más capacidad de sobrevivir que otras. Darwin afirmaba que la selección natural era lo más importante del proceso evolutivo de las especies.

3.1.3 Leyes de la herencia de Mendel

El monje austriaco Gregor Mendel realizó en 1856 una serie de experimentos sobre hibridación con plantas de chícharos o guisantes [51]. Durante el proceso de experimentación, Mendel notó de inmediato que al mezclar dos guisantes, un carácter[3] o variación propio de uno de los guisantes no aparecía en la próxima generación. Sin embargo, en la siguiente generación, al cruzar a los descendientes entre sí, este carácter aparecía de nuevo.

[1] Conocido actualmente como genotipo.
[2] Conocido actualmente como fenotipo.
[3] Conocido actualmente como alelo.

Estos experimentos lo condujeron a la definición de 3 leyes básicas que gobiernan el paso de una característica de un miembro de una especie a otra:

Ley de segregación Establece que los miembros de cada par de caracteres de una unidad hereditaria[4] se separan cuando se producen los gametos durante la meiosis, manifestándose sólo uno de ellos.

Ley de independencia Establece que las unidades de herencia se independizan durante la formación del gameto.

Ley de uniformidad Establece que cada característica heredada se determina mediante dos factores provenientes de ambos padres, lo que decide si una cierta unidad hereditaria es dominante o recesiva.

Mendel concluyó que no existen mezclas de unidades hereditarias como se creía hasta entonces, sino que sólo se combinan en la reproducción conservando su individualidad a través de las generaciones[5]. El resultado de estas combinaciones puede ser estimado estadísticamente.

3.1.4 Neo-Darwinismo

El conjunto de las teorías antes mencionadas sirven como base al paradigma llamado Neo-Darwinismo, que explica la evolución como un proceso de adaptación al medio ambiente y transferencia genética donde los individuos más aptos tienden a sobrevivir.

Las características del Neo-Darwinismo, según D. B. Fogel [27] son:

1. El individuo es el blanco principal de la selección.
2. La variación genética es en gran parte un fenómeno debido al azar. Los procesos estadísticos juegan un papel importante en la evolución.
3. La variación genotípica es en gran parte un producto de la recombinación y "sólo en segunda instancia la mutación".
4. La evolución gradual puede incorporar discontinuidades fenotípicas.
5. No todos los cambios fenotípicos son necesariamente consecuencia de una selección natural "ad-hoc".
6. La evolución es un cambio en la adaptación y la diversidad, y no simplemente un cambio en la frecuencia de los genes.

[4] Conocida actualmente como gene.

[5] Desafortunadamente el trabajo de Mendel nunca fue conocido por Darwin.

7. La selección es probabilística y no determinística.

Según el punto de vista de cada experto la evolución puede ser vista como un proceso de aprendizaje (W. D. Cannon [10]) o como un proceso de optimización (H. J. Bremermann [8, 32]).

3.2 Computación evolutiva

El Neo-Darwinismo sirve como inspiración para intentar simular dichos procesos evolutivos en una computadora. Estos algoritmos adquieren los elementos necesarios del proceso evolutivo natural [52]. Existen 5 elementos básicos para realizar su modelado:

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

La computación evolutiva es aplicada a varios problemas de la ciencia computacional e ingeniería tales como problemas de optimización y aprendizaje de máquina, que resultarían muy difíciles de resolver por métodos convencionales debido principalmente a su espacio de búsqueda extremadamente grande, complejo y con restricciones difíciles de satisfacer.

En las técnicas evolutivas se manipula un conjunto de soluciones potenciales en cada iteración, lo que implica un alto grado de paralelismo, pues se explotan a la vez diferentes regiones del espacio de búsqueda. Además los operadores probabilísticos evitan quedar atrapado en un óptimo local.

La inteligencia artificial considera a las técnicas evolutivas como heurísticas sub-simbólicas, debido a que su representación del conocimiento es numérica y no simbólica.

Existen distintas variantes de las técnicas evolutivas, pero se podrían agrupar en 4 principales paradigmas:

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

3.2.1 Programación evolutiva

A principios de los 1960's Lawrence J. Fogel propuso una técnica para resolver problemas de predicción de secuencias simulando los procesos evolutivos [29].

La técnica, llamada *Programación evolutiva* (PE) [30], consistía básicamente en generar una población de autómatas de estados finitos, expuestos a una serie de símbolos de entrada. Eventualmente, la técnica sería capaz de predecir las secuencias futuras de los símbolos que recibirían. A cada autómota se le asignaba un valor aptitud en relación a qué tan bueno era para predecir un símbolo. Los hijos eran generados por un mecanismo de copiado y se les aplicaba mutación con una distribución de probabilidad uniforme a estas copias. El operador de mutación consistía en generar cambios en las transiciones y en los estados de los autómatas (fenotipo). En una corrida típica, la mitad de la población con mayor aptitud se mantenía y el resto se sustituía por variantes de la mejor mitad.

El algoritmo básico de la PE es:

1. Generar una población inicial en forma aleatoria
2. Aplicar el operador de mutación. Cada padre genera un solo hijo
3. Calcular la aptitud de los individuos y realizar el proceso de selección
4. Reemplazar la población actual con la obtenida en la selección

La programación evolutiva es una abstracción de la evolución a nivel de las especies, por lo que no se considera el operador de recombinación sexual (cruza). Actualmente existen muchas variantes de esta técnica que contemplan otros métodos, como la selección por torneo o variantes de la mutación y otros dominios de aplicación [27, 28].

3.2.2 Estrategias evolutivas

Las *Estrategias evolutivas* (EE) fueron desarrolladas paralelamente a la PE por unos estudiante de doctorado en Alemania. Peter Bienert [6], Ingo Rechenberg [57] y Hans-Paul Schwefel [60] trabajaban en problemas de hidrodinámica en un túnel de viento los cuales requerían resolver un complejo problema de optimización. Ellos propusieron la idea de utilizar eventos estadísticos en una variable de control que produciría cambios aleatorios en una variable objetivo, lo que conduce a una mutación discreta de las variables objetivo. Esto permitía explorar complejos espacios de búsqueda.

La versión original $(1+1)-EE$ usaba un solo padre y con él se generaba un solo hijo. El hijo se mantenía si era mejor que el padre: de lo contrario se eliminaba (selección extintiva).

Un nuevo individuo se generaba usando:

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

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

El concepto de población fue introducido por Rechenberg [57], al proponer la estrategia $(\mu + 1) - EE$, en la cual hay μ padres y se genera un solo hijo, el cual puede reemplazar al peor padre de la población.

Schwefel [61] introdujo en 1975 el uso de múltiples hijos en las denominadas $(\mu + \lambda) - EE$ y $(\mu,\lambda) - EE$. La notación se refiere al mecanismo de selección utilizado:

- 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.

En las estrategias evolutivas no sólo se evolucionan las variables objeto sino también las variables de control (parámetros). Esto permite la autoadaptación. La cruza es un operador secundario y existen diferentes técnicas como la recombinación sexual (promedios ponderados) o la panmítica. Se utiliza por lo general una selección determinista.

En la programación evolutiva se trabaja a nivel de los individuos y directamente sobre el fenotipo, ya que no existe una codificación de las variables.

3.2.3 Algoritmos genéticos

Uno de los paradigmas más populares es el *Algoritmo genético* (AG), desarrollado por Holland y sus estudiantes en la universidad de Michigan [35] a principios de los 1960s.

El AG tiene dos principales características: Utiliza una representación de cadena fija binaria (cromosoma) y hace un alto uso de la recombinación. La representación de individuos con cadenas fijas de ceros y unos (genotipo) implica encontrar un mapeo adecuado de las variables. Actualmente se cuenta con diferentes representaciónes (codificaciones) además de la binaria [15]. Al igual que las EE, los AG realizan los procesos evolutivos a nivel de los individuos.

El algoritmo básico de un AG simple es el siguiente [9]:

1. Generar aleatoriamente una población inicial de cromosomas
2. Calcular la aptitud de cada individuo
3. Seleccionar (por lo general en forma probabilística) a los individuos con base en su aptitud
4. Aplicar los operadores genéticos de cruza y mutación para generar la siguiente población
5. Reemplazar la población actual con la obtenida en la selección
6. Ciclar hasta que cierta condición se satisfaga

Como se mencionó anteriormente, los individuos son representados por cadenas de longitud fija llamadas cromosomas y cada posición dentro del cromosoma es conocida como gene. Cada uno de los genes pueden tomar un valor indistinto dependiendo de la representación que se maneje. A esto se le llama alelo.

Existen distintas formas de realizar el proceso de selección en los AG:

- *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, entre otras [33].
- *Selección Mediante Torneo* propuesta por Wetzel [69]. Los individuos son seleccionados con base en comparaciones directas. Existen dos variantes: la determinista y la estocástica.

- *Selección de estado Uniforme* propuesta por Whitley[71]. Se usa en algoritmos genéticos no generacionales, en los cuales sólo unos cuantos individuos son reemplazados en cada generación.

Las técnicas de cruza permiten una alta recombinación de material cromosómico. Existen principalmente tres tipos de cruza:

- *Cruza de un punto:* se selecciona un punto de manera aleatoria dentro del cromosoma de cada padre, y a partir de éste se intercambian los materiales genéticos, para dar origen a nuevos descendientes [35].

- *Cruza de dos puntos:* esta técnica es similar a la anterior, pero se generan dos puntos de cruza por cada padre [39].

- *Cruza Uniforme:* consiste en una cruza de n puntos, pero el número de puntos de cruza nunca se fija previamente [24, 64].

La mutación es un operador secundario en los AG que permite realizar pequeños saltos en el espacio de búsqueda. Por lo general consiste en un operador not (binario), pero éste varía según la representación que se utilice.

3.2.4 Programación genética

La programación genética es un paradigma propuesto por dos investigadores N. L. Cramer [22] y J. Koza [44], quienes sugirieron que una estructura de árbol podía ser usada como la representación de un programa en un genoma. Sin embargo J. Koza fue el primero en reconocer la importancia de esta técnica y demostró su factibilidad para la programación automática en general [45].

Los individuos en la programación genética son programas de computadora estructurados jerárquicamente. El tamaño, la forma y el contenido de estos programas de computadora pueden cambiar dramáticamente durante el proceso de evolución. Los individuos están formados por un conjunto de términos y funciones, que actúan como primitivas que sirven de base para la construcción de programas.

El **conjunto de términos** se compone por todas aquellas variables, constantes o funciones de aridad cero, que sirven como argumentos para las funciones. En una estructura de árbol los términos son considerados como las hojas de las ramas o nodos tipo término.

El **conjunto de funciones** se compone por todos aquellos estatutos, operadores aritméticos, operadores binarios o funciones de dominio específico.

Por ejemplo el conjunto de funciones binarias {AND, OR, NOT, XOR}. En una estructura de árbol se conocen como nodos tipo función.

Existen algunas propiedades que deben ser consideradas al construir el conjunto de términos y funciones:

1. Suficiencia: Las funciones y términos usados deben de ser lo bastante capaces como para representar una solución al problema.

2. *"Closure"*: Cada función debe ser capaz de manejar adecuadamente todos los valores que pudiera recibir como entrada. Por ejemplo, contemplar una función de división entre cero cuando sea necesaria.

3. Universalidad: El conjunto de funciones y términos debe ser práctico y útil. Es decir, no debe perderse tiempo diseñando funciones y términos que parecen perfectamente atenuados al problema.

Los programas (individuos) son estructuras compuestas de funciones y términos unidos con reglas o convenciones de cuándo y cómo cada función o término será ejecutado. La elección de una estructura de programa en la PG afecta el orden de ejecución, el uso de la memoria, y la aplicación de los operadores genéticos al programa. Existen 3 principales estructuras de programa en la PG [5, 48]:

- Estructuras de árbol: es una estructura sencilla de árbol, el cual puede ser recorrido en forma postfija o prefija (Figura 3.1).

- Estructuras lineales: es una cadena simple de instrucciones, que puede ser ejecutada según una interpretación definida. Una de las más populares son las expresiones canónicas (Expresiones-S).

- Estructuras de grafo: son estructuras complejas de grafos compuestas por nodos y aristas. Se utilizan para modelar problemas mucho más complejos.

Al inicializar la población se debe tomar en cuenta la profundidad o tamaño que tendrán los programas. El parámetro de profundidad es crucial ya que permite controlar la complejidad (número de nodos o instrucciones) que tendrá el programa, dependiendo de la estructura que se utilice.

Los métodos de selección utilizados en la PG son los mismos que los utilizados en los AG mencionados en la sección anterior.

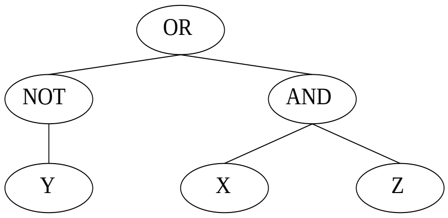

Figura 3.1: Representación arbórea de un individuo en PG equivalente a la Expresión-S (OR (NOT Y) (AND X Z)).

El operador de cruza combina el material genético de dos padres, intercambiando parte de un padre con una parte del otro[6] como se muestra en la Figura 3.2. La cruza consiste en [45, 5]:

- 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

Existen 4 operadores secundarios (asexuales) en la PG. Estos son aplicados en bajos porcentajes a la población:

1. *Mutación:* Selecciona un punto al azar en la estructura y se reemplaza el subárbol con uno nuevo generado aleatoriamente (Figura 3.3)
2. *Permutación:* Selecciona un punto al azar y se reordenan los argumentos que se desprendan del nodo función seleccionado
3. *Edición:* Selecciona un punto al azar y reemplaza el subárbol de acuerdo a un conjunto de reglas, obteniendo un nuevo subárbol o un término. Por ejemplo: (AND X X) = X
4. *Encapsulamiento:* Consiste en la identificación automática de subárboles potencialmente útiles que posteriormente son referenciados [46]

El algoritmo básico para la PG [5] es el siguiente:

1. Inicializar la población

[6]Generalmente, los subárboles en cada padre difieren uno de otro en su contenido y/o tamaño

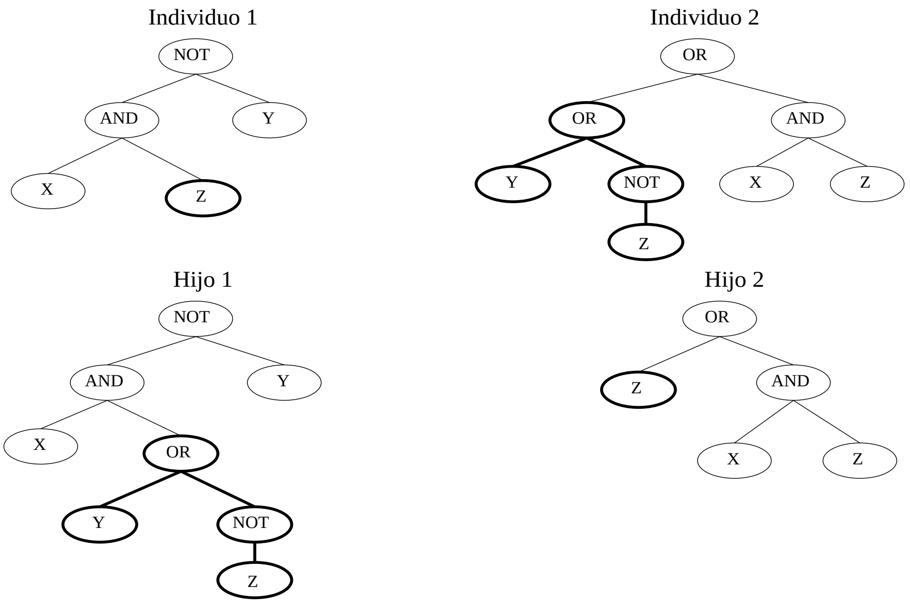

Figura 3.2: Mecanismo de cruza de dos estructuras arbóreas que generan 2 nuevos hijos.

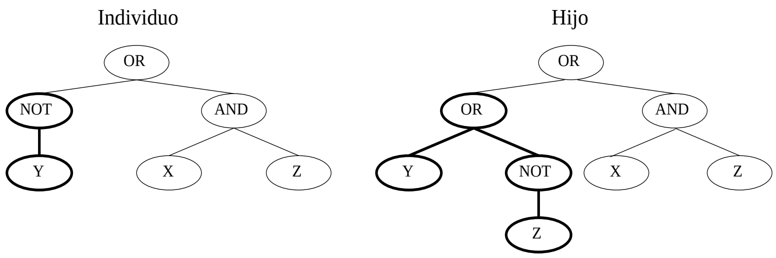

Figura 3.3: Mecanismo de mutación que genera nuevos subárboles.

2. Evaluar los programas en la población existente y asignar un valor de aptitud a cada individuo
3. 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 a la nueva población
4. Reemplaza la población existente con la nueva población, hasta cumplir el criterio de terminación
5. Presentar al mejor individuo de la población.

3.3 Hardware evolutivo

Tradicionalmente los sistemas electrónicos han sido diseñados por ingenieros utilizando diversas técnicas, reglas y principios de diseño, así como su experiencia. El *Hardware Evolutivo* consiste en el diseño de circuitos electrónicos que son evolucionados a través de un proceso de selección natural [53].

George J. Friedman [31] fue uno de los primeros en aplicar técnicas evolutivas a la robótica. Friedman propuso un mecanismo para construir, probar y evaluar circuitos en forma automática, utilizando mutaciones aleatorias y procesos de selección. Este sería probablemente el primer trabajo de *Hardware Evolutivo.*

La idea principal consiste en codificar los circuitos en un cromosoma y utilizar un proceso de ensamble y prueba, que unido a un proceso evolutivo nos permita diseñar circuitos de distinto grado de complejidad y así explorar de manera más eficiente el espacio de diseño.

Existen procesos de evolución dirigidos a *nivel de compuertas* que consisten en emplear sólo compuertas lógicas básicas como pueden ser AND, OR y NOT. Y otro dirigido a *nivel de funciones* que consiste básicamente en utilizar compuertas lógicas y a su ves con ellas construir nuevos módulos compuestos, como las ADF's entre otras [37, 53, 40].

Actualmente el hardware evolutivo también se divide de acuerdo al proceso de evaluación [40]:

- Extrínseca: las soluciones son construidas y evaluadas como modelos (simulaciones)

- Intrínseca: las soluciones son implementadas en dispositivos reconfigurables los cuales son evaluados con algunos equipos
- Mixtrínseca: es una población compuesta de modelos y dispositivos reconfigurables

Se cuenta con implementaciones en Algoritmos Genéticos [14, 11, 12, 19] y Sistema de Hormigas [20, 21] que emplean una representación matricial del circuito y otras en Programación Genética que utilizan distintas representaciones [45, 53, 40, 67]. Algunas de estas técnicas por lo general necesitan un mapeo del cromosoma para evaluar algún circuito.

En este trabajo de tesis proponemos el uso de la programación genética con una representación lineal en prefijo que permita diseñar (a nivel de compuertas), probar y evolucionar expresiones lógicas que finalmente produzcan un circuito lógico.

Capítulo 4

Descripción de la técnica

A continuación se describe el planteamiento para la solución del diseño de circuitos lógicos combinatorios utilizando los beneficios de la Programación Genética (PG). Como sabemos, la PG utiliza una representación arbórea de expresiones, también llamada expresiones canónicas ó expresiones S en Lisp [45].

En este capítulo discutiremos la manera en que se decidió implementar esta representación arbórea, así como la forma en que las expresiones booleanas son evaluadas. Adicionalmente, se describe la función de aptitud y los operadores genéticos adoptados. Finalizamos este capítulo con una descripción paso a paso de un ejemplo de uso de la implementación.

4.1 Representación de los individuos

Cualquier circuito lógico puede ser representado a través de funciones booleanas que generalmente se escriben como expresiones infijas de la siguiente manera:

$$F = X' \cdot Y$$

La forma gráfica de representar las expresiones es por medio de un árbol de evaluación como muestra la Figura 4.1. La representación gráfica de la expresión resulta clara y define perfectamente las partes en que se divide la expresión y recalca la importancia de la última operación a realizar (nodo raíz).

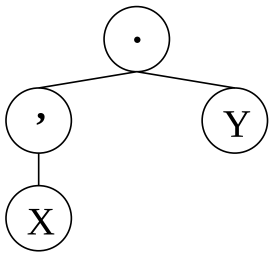

Figura 4.1: Representación gráfica de la expresión $F = X' \cdot Y$ en un árbol de evaluación. El nodo "·" es considerado el nodo raíz.

Esta representación infija es fácil de leer para un diseñador humano, no así para una computadora. Los compiladores comunes utilizan otras representaciones para poder leer las expresiones y evitar conflictos con el orden de los operadores.

Las expresiones prefijas y postfijas[1] son las alternativas para representar expresiones infijas como se muestra a continuación:

1. En las expresiones prefijas el operador precede a los operandos. Esto significa que el árbol de evaluación es recorrido en preorden.

$$\text{Expresión prefija: } F = \cdot' XY$$

2. En las expresiones postfijas el operador va después de los operandos. Esto implica un recorrido del árbol de evaluación en posorden.

$$\text{Expresión postfija: } F = X'Y\cdot$$

La PG utiliza diversas formas de representación de individuos (*genomas*) [5]. Una representación lineal en forma prefija luce como la opción más adecuada para solucionar el problema de diseño de circuitos lógicos [42], debido a su sencillez implícita de evaluación con un intérprete de genomas que será discutido más adelante.

El genoma de cada individuo tiene una distribución de cadena prefija constituída por un conjunto de funciones y términos:

genoma: &X^|!ZWY

[1] Los prefijos "pre" y "post" se refieren a la posición relativa del operador respecto a los operandos.

El conjunto de *funciones* está formado por caracteres que representan una función lógica determinada como se muestra en la Tabla 4.1.

Compuerta	Símbolo	Aridad
NOT	!	1
AND	&	2
XOR	ˆ	2
OR	|	2

Tabla 4.1: Conjunto de funciones utilizadas en la representación de circuitos lógicos en Programación Genética.

El conjunto de *términos* son carácteres de la A a la Z que tienen aridad cero y se colocan en la parte inferiror de cada rama (hojas) como lo muestra la Figura 4.2.

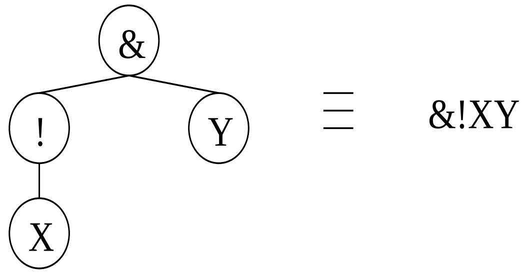

Figura 4.2: A la izquierda una expresión representada en un árbol. A la derecha la expresión prefija equivalente

Nuestro mecanismo de generación de individuos asegura producir expresiones prefijas válidas, que no necesitan verificaciones posteriores. El número de nodos tipo función y término que componen una expresión prefija válida varía según la complejidad del circuito. En todos los experimentos realizados se utilizaron cadenas de 2^7 (128) nodos. El tamaño de las cadenas es controlado en los procesos de cruza y mutación. Esto nos permite evitar posibles problemas al momento de la evaluación, evitando errores de desborde de memoria en la interpretación de la expresión.

Finalmente, todos los árboles iniciales que componen la población, cuentan con las características de ser un 50% completos (*"full"*), es decir, gráficamente balanceados. El otro 50% son árboles que varían en su forma (*"grow"*). Esta característica nos permite tener una población más robusta (variada) de expresiones [45].

4.2 Evaluador de expresiones

Las cadenas son evaluadas por un *intérprete de genomas* (parser). El intérprete lee las expresiones carácter por carácter y las evalúa según su significado (ver Figura 4.3). En tal caso:

1. Si el carácter extraído es una letra, se le asignará un valor del tipo tabla de verdad[2], que corresponderá a las combinaciones binarias que se hayan establecido para esa variable en especial. Este valor es insertado en una pila tipo tabla de verdad. Cada nivel en la pila es de 2^n posiciones, donde n es el número de variables de entrada para el circuito.

2. En caso de no ser una letra, se trata de un operador lógico. En este caso, los valores son extraídos de la pila, evaluados por el operador lógico y el resultado es depositado nuevamente en la parte superior de la pila. Este procedimiento continúa hasta que se haya terminado de evaluar toda la expresión.

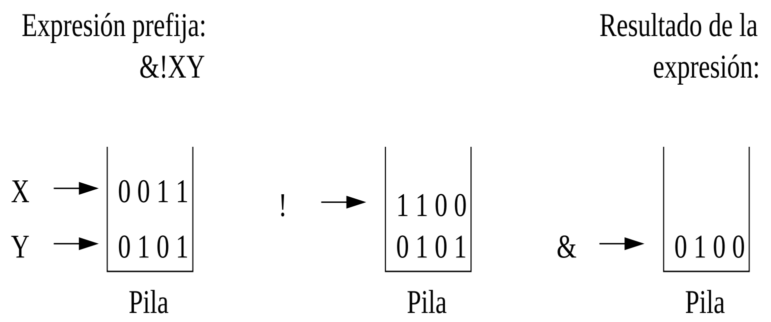

Figura 4.3: Se muestra el funcionamiento del evaluador de expresiones (parser). Vemos cómo lee carácter por carácter y como va solucionando el resultado de la expresión lógica.

El resultado obtenido tras la evaluación de la expresión, está compuesto por una variable del tipo tabla de verdad de 2^n posiciones, donde n es el número de variables de entrada para el circuito a diseñar.

[2]El tipo tabla de verdad es un vector de 2^n posiciones compuestas por ceros y unos.

4.3 Función de aptitud

Posteriormente, el resultado lógico de la expresión es comparado bit a bit con la expresión de salida deseada del circuito, a fin de determinar su valor de aptitud. A mayor número de ocurrencias o aciertos con respecto de la salida deseada, mayor será la aptitud del individuo. La función de aptitud es mostrada a continuación:

$$aptitud(i) = aciertos(i)/2^n$$

La aptitud del *i-ésimo* individuo es igual al número de aciertos o similitudes con cada uno de los bits de salida del circuito deseado. El total de aciertos es divido entre 2 elevado a la n variables de entrada del circuito.

4.4 Operadores genéticos

La **reproducción** es implementada utilizando un método de selección proporcional llamado Selección Universal Estocástica [3]. Esta técnica se usa debido a que buscamos minimizar la mala distribución de los individuos en la población en función de sus valores esperados (la aptitud del individuo dividida entre la aptitud promedio de la población).

El operador de **cruza** consiste en seleccionar dos individuos. Después, se generan puntos de cruza al azar dentro de cada cadena de genoma de manera independiente y se determina el tamaño de cada una de las subcadenas válidas que serán intercambiadas. Si alguna subcadena a ser insertada provoca un crecimiento excesivo, se repite el proceso hasta que se logre la inserción de una subcadena de tamaño apropiado. Finalmente se construyen los nuevos genomas intercambiando las subcadenas de cada individuo como se muestra a continuación:

$$genoma1 = Inicio\_de\_Padre1 + Subcadena2 + Resto\_Padre1$$
$$genoma2 = Inicio\_de\_Padre2 + Subcadena1 + Resto\_Padre2$$

Las combinaciones de cruza que se producen varían según la complejidad de las cadenas y de los puntos de cruza seleccionados. Por ejemplo, puede existir el caso de intercambiar toda una cadena de genoma por una subcadena. Los descendientes varian en complejidad y en número de componentes, de esta forma distan de ser iguales a sus padres.

La **mutación** es el operador genético que nos permite explorar algunos puntos del espacio de diseño. En la mutación seleccionamos un individuo y escogemos un punto dentro de la cadena. Determinamos el tamaño de la subcadena que será removida de la cadena y generamos aleatoriamente una nueva expresión prefija que será insertada.

Finalmente se realiza un proceso de **elitismo** que asegure la transferencia genética del mejor individuo en esa población. En este caso se inserta en la siguiente generación al individuo que obtenga la aptitud más alta y cuya longitud de cadena sea la menor.

4.5 Un ejemplo

Para explicar en forma más detallada el procedimiento de la técnica utilizada denominada Programación Genética Prefija (PG Prefija), tomaremos el ejemplo de un circuito *"4-even parity"*[3]. El número de variables de entrada utilizadas para este circuito es de 4, identificadas por las variables lógicas Z, W, X y Y. La salida del circuito que deseamos encontrar en los individuos es identificada por F. La Tabla 4.2 muestra la configuración de la tabla de verdad para solucionar el circuito.

Z	W	X	Y	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Tabla 4.2: Tabla de verdad para el circuito 4-even parity.

[3]Este y todos los experimentos se realizaron en un sistema PentiumII 300MHz, 128MB en RAM y Red Hat Linux 6.1. El sistema PG Prefija fue compilado en GNU CC v2.0.

El experimento se realizó con los siguientes parámetros: un tamaño de población de 90 individuos, un número máximo de generaciones de 120, un porcentaje de cruza del 95% y un porcentaje de mutación del 5%.

Las expresiones prefijas que componen la población inicial pueden variar en forma y tamaño ya que son producidas en forma aleatoria. A continuación presentamos algunos ejemplos:

genoma1: &|X!YW
genoma2: ^Y||&ZXWZ
genoma3: !^&XW!Z

El evaluador de expresiones toma las cadenas y las interpreta valiéndose de una pila según correspondan los valores de cada carácter, el resultado es un vector que contiene la configuración de la salida lógica.

La función de aptitud es calculada según el número de similitudes de la expresión evaluada con respecto a la salida deseada del circuito y el resultado es dividido entre el número total de posiciones en el vector que es 2^n, donde n es el número total de variables, como se muestra en el siguiente ejemplo (expresión evaluada genoma1):

salida deseada "F"	1001011001101001
genoma1 evaluado	0000101100001011

En este caso se tienen 8 similitudes del genoma1 con respecto a la salida F, entonces la aptitud es:

$$aptitud(genoma1) = aciertos(genoma1)/2^4 = 8/16 = 0.50$$

La reproducción es realizada con una Selección Universal Estocástica, seleccionando el mayor número posible de individuos aptos (mayor aptitud).

La cruza se realiza en dos individuos, seleccionando dentro de sus cadenas subcadenas e intercambiándolas de un individuo a otro como enseguida se muestra.

Cadenas padre y **subcadenas** seleccionadas para el cruzamiento:

genoma1: &|X! **Y** W
genoma2: ^Y|| **&ZX** WZ

Cadenas hijo que resultan de la cruza:

genoma1: &|X! **&ZX** W
genoma2: ^Y|| **Y** WZ

Para la mutación se selecciona un individuo y en su cadena se escoge un punto al azar y se extrae la subcadena resultante la cual será sustituída por una nueva expresión.

Individuo y **subcadena** seleccionada para la mutación:

genoma3: !^&XW! **Z**

Expresión generada aleatoriamente: **&&!XWY**

Individuo resultante de la mutación:

genoma3: !^&XW! **&&!XWY**

El elitismo selecciona al individuo cuya aptitud sea la mayor y de longitud de cadena menor. Para este ejemplo en la generación 7 se produce un genoma que representa a un circuito 100% funcional.

genoma: ^^^&X|XY^Z|Y!!Y^!ZWZ
resultado = 1001011001101001
número de nodos = 20
aptitud = 1.0000

En la generación 10 se produce una expresión de menor tamaño. Al verificarla encontramos una simplificación booleana[4], ya que $X(X+Y) = X$, lo que implica remover la subcadena &X|XY.

genoma: ^^^X^Z|Y!!Y^!ZWZ
resultado = 1001011001101001
número de nodos = 16
aptitud = 1.0000

En la generación 12 se logra eliminar la doble negación de la subcadena !!Y, que en el álgebra booleana es $Y'' = Y$, produciéndose un resultado más corto.

genoma: ^^^X^Z|YY^!ZWZ
resultado = 1001011001101001
número de nodos = 14
aptitud = 1.0000

[4] En la programación genética existe un operador de *Edición* que permite la simplificacion de este tipo, aunque en esta implementación no fue incluido.

En la generación 14 se produce la simplificación booleana de $Y + Y = Y$ de la subcadena |YY.

genoma: ^^^X^ZY^!ZWZ
resultado = 1001011001101001
número de nodos = 12
aptitud = 1.0000

Y finalmente, en la generación 19 se encuentra una expresión más corta, la cual se compone de 4 operaciones lógicas 3 XORs y 1 NOT que son consideradas finalmente como compuertas.

genoma ^X^^!ZWY
resultado = 1001011001101001
número de nodos = 8
aptitud = 1.0000

Resulta interesante explicar lo ocurrido entre el resultado de la generación 14 y 19. Aquí se produce algo conocido por los diseñadores de circuitos como *"Buffer"*, es decir, lo que entra sale. Resulta más práctico explicarlo en forma algebraica. A continuación desarrollamos en forma infija las expresiones de la generación 14 y 19, que se muestran a continuación:

14) $X \oplus Z \oplus Y \oplus Z' \oplus W \oplus Z$
19) $X \oplus Z' \oplus W \oplus Y$

Aplicando las leyes distributiva y asociativa a la expresión 14, obtenemos:

$$14a)\ (X \oplus Z' \oplus W \oplus Y) \oplus (Z \oplus Z)$$

El resultado de $(Z \oplus Z) = 0$, para un diseñador humano es interpretado como tierra, es decir sin corriente. Además cualquier salida de circuito con XOR a tierra es interpretado como un seguidor de paso o *"Buffer"*. Es decir, el resultado no cambia al pasar por la XOR. Esto nos conduce a la eliminación de la parte derecha de la expresión 14a, obteniendo el resultado de la expresión 19.

La expresión continúa constante de la generación 19 en adelante. El valor de aptitud ha llegado a su punto máximo y el proceso de elitismo mantiene el resultado hasta el final ya que se produce una salida factible lo bastante corta para ser tomada como una buena solución.

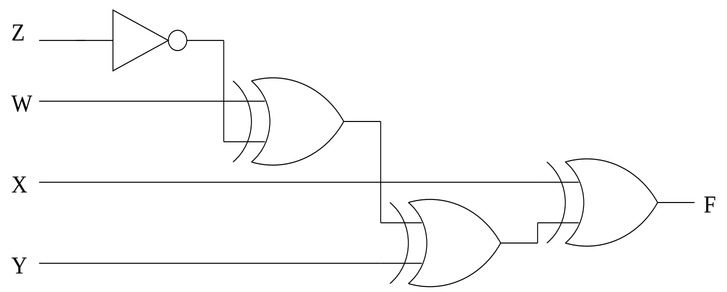

Figura 4.4: Circuito desarrollado a partir de la expresión $a)$ con 4 niveles.

La solución obtenida puede ser fácilmente interpretada como una expresión infija y manipularse algebraicamente, encontrando distintas equivalencias para la misma expresión. Para este ejemplo encontramos las siguientes equivalencias:

1. La expresión $a)$ es una posible solución, si la interpretámos en forma prefija tal cual. Esta expresión nos produce un circuito de 4 niveles como lo muestra el diagrama lógico de la Figura 4.4.

$$a)\ F = X \oplus ((Z' \oplus W) \oplus Y)$$

2. Analizando algebraicamente el resultado anterior y aplicando las leyes distributiva y asociativa obtenemos la siguiente expresión:

$$b)\ F = (Z' \oplus W) \oplus (X \oplus Y)$$

   El número de niveles que produce la expresión $b)$ es de 3, lo que implica un menor retardo en la señal emitida a través del circuito. Además, resulta más sencillo de entender gráficamente como se observa en el diagrama lógico de la Figura 4.5.

Con la misma expresión resultante desarrollamos dos posibles soluciones. Después de un análisis algebraico llegamos a la conclusión de que el circuito $b)$ de tres niveles es la mejor solución al problema. La Figura 4.6 muestra el comportamiento de los valores obtenidos del ejemplo.

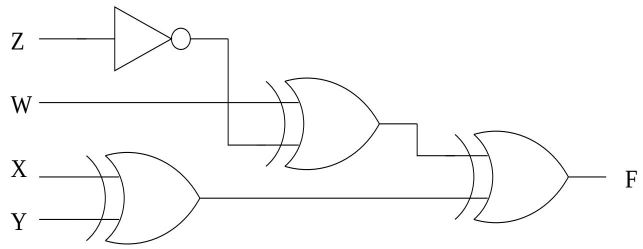

Figura 4.5: Circuito desarrollado a partir de la expresion *b*) con 3 niveles.

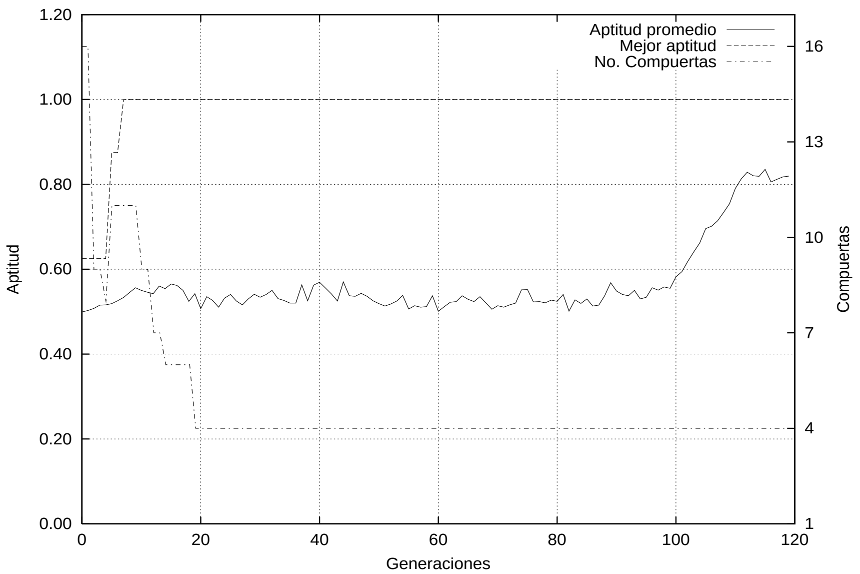

Figura 4.6: Comportamiento de la función aptitud promedio, la aptitud del mejor individuo y su número de compuertas a lo largo de la corrida.

Capítulo 5

Circuitos de una salida

5.1 Experimentos

Los circuitos expuestos en este capítulo constan de varias entradas y una sola salida. Se seleccionaron 5 circuitos lógicos de distinto grado de complejidad para probar la técnica propuesta para diseñar circuitos lógicos, denominada Programación Genética Prefija (PG Prefija). Los resultados son comparados con aquellos obtenidos por otras técnicas tradicionales como el método gráfico de Mapas de Karnaugh [41] y el método tabular de Quine-McCluskey [56, 50]. Ambos son métodos muy utilizados por diseñadores humanos. También se comparará un resultado contra el método de Sasao [59], que realiza el proceso de diseño con uso exclusivo de compuertas ANDs y XORs.

Asimismo, se compará también la efectividad de la PG Prefija contra otras técnicas evolutivas como son el Algoritmo Genético Multiobjetivo (MGA) [19, 18], el Algoritmo Genético Binario (BGA) [11, 13] y el Algoritmo Genético de cardinalidad-N (NGA) [12, 17, 16], así como contra una implementación basada en la meta-heurística denominada Colonia de Hormigas [26, 25], a la cual llamaremos Sistema de Hormigas (AS) [20, 21].

Para cada circuito se efectuaron 20 corridas en forma aleatoria, con el fin de realizar un análisis y medir su efectividad generando circuitos factibles. El número de generaciones y el tamaño de la población varía según la complejidad del circuito. Cada individuo de nuestra implementación de PG Prefija consta de una profundidad máxima de 128 nodos. Después de varias pruebas se decidió utilizar un porcentaje de cruza de 95% y un porcentaje de mutación de 5%.

X	Y	Z	F
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 5.1: Tabla de verdad para el circuito del primer ejemplo.

5.1.1 Ejemplo 1

El primer ejemplo consiste en un circuito de 3 entradas y 1 salida, con la tabla de verdad que se muestra en la Tabla 5.1. Se realizaron 20 corridas con un tamaño total de población de 150 individuos y un número máximo de 100 generaciones. Los resultados se muestran en la Tabla 5.2. El 100% de las corridas produjo circuitos factibles y el 45% de las ocasiones arrojó soluciones óptimas[1] con 4 compuertas. La corrida 8 con 6 compuertas se encuentra cercana a la mediana y describe mejor el comportamiento de la PG Prefija en estos experimentos. Su desempeño es mostrado en la Figura 5.1. La aptitud promedio de las 20 corridas nos da como resultado una media de 0.7677, con una desviación estándar de 0.0328 y la mediana en 0.7621.

La mejor solución ocurrió en la corrida 9, alcanzando la zona factible en la generación 24 con 6 compuertas y encontrando el óptimo en la generación 82 con 4 compuertas. El diagrama lógico del circuito óptimo es mostrado en la Figura 5.3. La peor solución obtenida tuvo 11 compuertas y ocurrió en la corrida 4.

Este circuito es pequeño y nos permite analizar un detalle interesante. Como se explicó en el capítulo anterior, la función de aptitud consiste sólo de un objetivo: maximizar el número de aciertos con respecto a la salida deseada del circuito. Se cuenta con un proceso de elitismo que nos permite conservar aquellos circuitos que sean de menor tamaño, lo que nos lleva a soluciones factibles las cuales son, adicionalmente, óptimas.

[1]Usaremos el término “óptimo” para referirnos a las mejores soluciones conocidas para el circuito, aunque el óptimo global es, en términos generales, desconocido para un circuito combinatorio cualquiera.

No. Corrida	Aptitud	Aptitud Promedio	No. Compuertas
1	1.0000	0.8235	4
2	1.0000	0.7756	5
3	1.0000	0.7576	4
4	1.0000	0.7505	11
5	1.0000	0.7953	5
6	1.0000	0.7770	5
7	1.0000	0.7415	9
8	1.0000	0.7647	6
9	1.0000	0.7476	4
10	1.0000	0.7303	4
11	1.0000	0.8239	4
12	1.0000	0.7447	4
13	1.0000	0.7138	5
14	1.0000	0.7231	4
15	1.0000	0.7990	5
16	1.0000	0.7974	6
17	1.0000	0.8171	5
18	1.0000	0.7594	7
19	1.0000	0.7685	4
20	1.0000	0.7438	4

Tabla 5.2: Análisis de 20 corridas para el primer ejemplo.

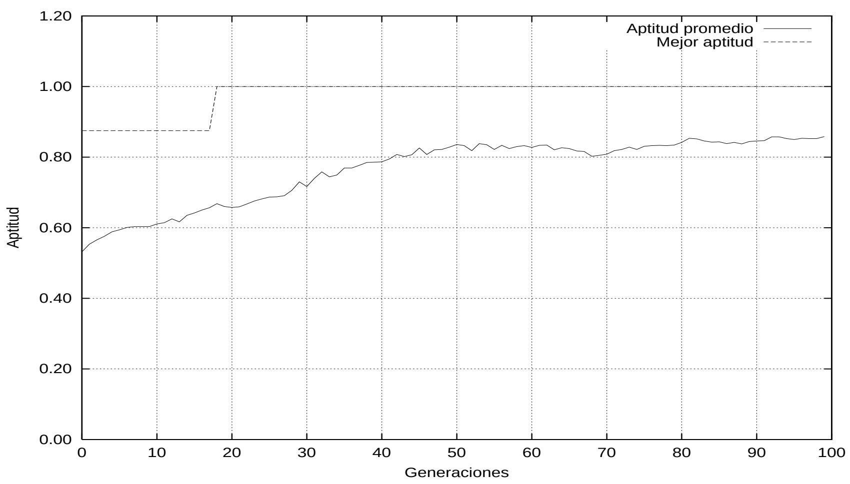

Figura 5.1: Gráfica de la corrida ubicada en la mediana del primer ejemplo.

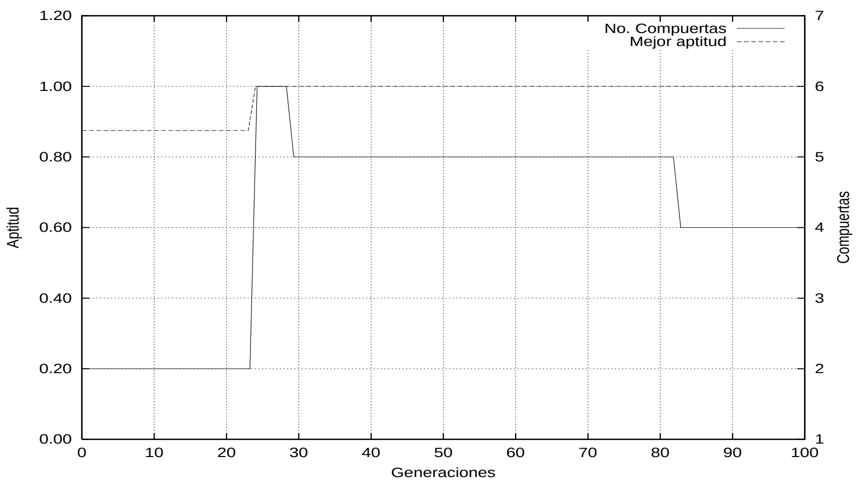

Figura 5.2: Comportamiento de la función de aptitud en relación al elitismo en la mejor solución obtenida para el circuito del primer ejemplo.

Desafortunadamente nunca será perceptible el impacto del elitismo, excepto al terminar la corrida. Para mostrar este impacto, veamos la Figura 5.2 que corresponde a la mejor solución del primer ejemplo. Observamos cómo la aptitud se conserva estable en 1.0000 y sin embargo el número de compuertas se reduce, por efecto del elitismo, hasta alcanzar el óptimo.

El resultado es comparado en la Tabla 5.3. Se tiene el Diseñador Humano 1, el cual utiliza Mapas de Karnaugh y álgebra booleana para simplificar. El Diseñador Humano 2 utiliza el procedimiento de Quine-McCluskey. Ambos obtienen resultados factibles pero no pueden llegar a una simplificación equiparable a la de la PG Prefija.

Finalmente, tenemos el MGA, el cual obtiene una solución factible de 4 compuertas. Observamos que ambos procedimientos evolutivos llegan a soluciones factibles. Aún cuando difieren en sus expresiones booleanas, los circuitos obtenidos son muy similares en el tipo y organización de las compuertas. La factibilidad de ambas técnicas es de 100%, aunque la PG Prefija utilizó un número de 15,000 iteraciones, en contraste a las 27,000 iteraciones del MGA.

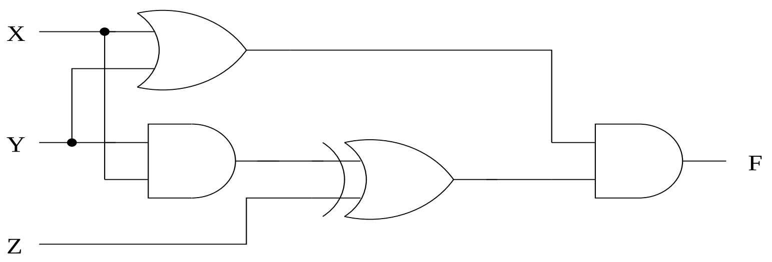

Figura 5.3: Diagrama lógico del mejor circuito producido por la PG Prefija para el primer ejemplo.

Diseñador Humano 1	Diseñador Humano 2
$F = Z(X \oplus Y) + Y(X \oplus Z)$	$F = X'YZ + X(Y \oplus Z)$
5 compuertas	6 compuertas
2 ANDs, 1 OR, 2 XORs	3 ANDs, 1 OR, 1 XOR, 1 NOT
PG Prefija	MGA
$F = (X + Y)((YX) \oplus Z)$	$F = (X + Y)Z \oplus (XY)$
4 compuertas	4 compuertas
2 ANDs, 1 OR, 1 XORs	2 ANDs, 1 OR, 1 XOR

Tabla 5.3: Comparación de las mejores soluciones obtenidas por la PG Prefija, el MGA, y Diseñadores Humanos para el circuito del primer ejemplo.

Z	W	X	Y	F
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
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 5.4: Tabla de verdad para el circuito del segundo ejemplo.

5.1.2 Ejemplo 2

En el segundo problema incrementamos la complejidad. Ahora contamos con un circuito de 4 entradas y 1 salida, cuya tabla de verdad se muestra en la Tabla 5.4. Se realizaron 20 corridas con un tamaño máximo de población de 400 individuos y un número máximo de 1000 generaciones. En un 90% de las corridas se obtuvieron circuitos factibles y en un 15% de ellas, se obtuvieron circuitos óptimos tal y como se muestra en la Tabla 5.5. La corrida 12, como lo vemos en la Figura 5.4, describe mejor el comportamiento de la PG Prefija, ya que se encuentra cercana a la mediana. La aptitud promedio de las 20 corridas nos da como resultado una media de 0.8121, con una desviación estándar de 0.0181 y la mediana en 0.8112.

La mejor solución apareció en la corrida 7, alcanzando la zona factible en la generación 47 con 72 compuertas y encontrando el óptimo en la generación 165 con 7 compuertas. El diagrama lógico del circuito óptimo es mostrado en la Figura 5.5. La peor solución obtenida tuvo 13 compuertas y ocurrió en la corrida 18.

No. Corrida	Aptitud	Aptitud Promedio	No. Compuertas
1	0.9375	0.8116	7
2	1.0000	0.8253	7
3	1.0000	0.8109	8
4	1.0000	0.8096	10
5	1.0000	0.8232	9
6	1.0000	0.8115	10
7	1.0000	0.8469	7
8	1.0000	0.7889	8
9	1.0000	0.8193	10
10	1.0000	0.7981	11
11	1.0000	0.8001	13
12	1.0000	0.8114	8
13	1.0000	0.8246	9
14	1.0000	0.8058	11
15	1.0000	0.8207	10
16	1.0000	0.8057	11
17	0.9375	0.793	11
18	1.0000	0.7839	13
19	1.0000	0.8568	7
20	1.0000	0.795	8

Tabla 5.5: Análisis de 20 corridas para el segundo ejemplo.

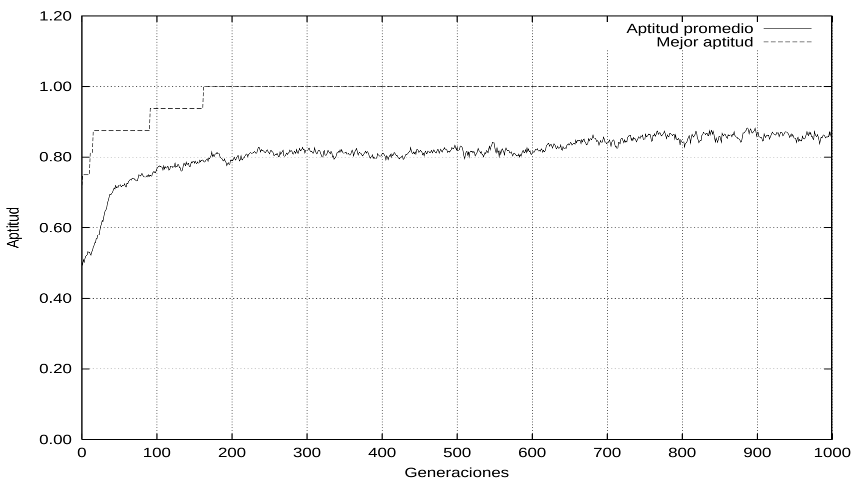

Figura 5.4: Gráfica de la corrida ubicada en la mediana del segundo ejemplo.

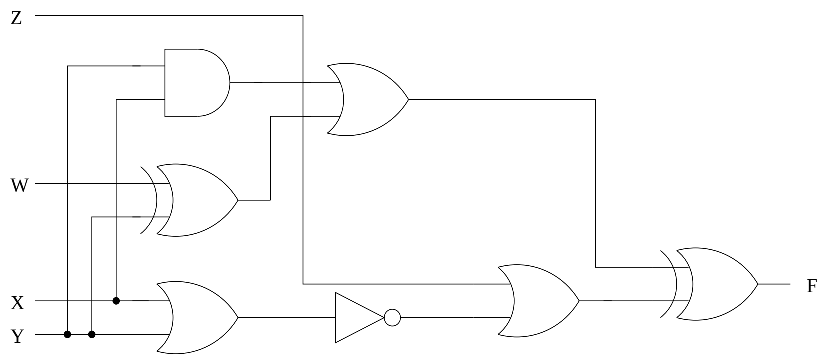

Figura 5.5: Diagrama lógico para el circuito óptimo producido por la PG Prefija para el segundo ejemplo.

La Tabla 5.6 muestra las soluciones obtenidas por otros métodos tradicionales y mediante técnicas evolutivas. El Diseñador Humano utiliza Mapas de Karnaugh. La técnica de Sasao ha sido utilizada para solucionar este circuito encontrando una configuración de 12 compuertas.

Sin embargo, ninguno de estos métodos llega a producir un circuito con menos compuertas que las técnicas evolutivas. El MGA llega a optimizar el circuito, al igual que la PG Prefija, con 7 compuertas, aunque esta última utiliza un mayor número de iteraciones: 400,000 contra las 68,000 iteraciones del MGA.

Diseñador Humano
$F = ((Z'X) \oplus (Y'W')) + ((X'Y)(Z \oplus W'))$
11 compuertas
4 ANDs, 1 OR, 2 XORs, 4 NOTs
Sasao
$F = X' \oplus Y'W' \oplus XY'Z' \oplus X'Y'W$
12 compuertas
5 ANDs, 3 XORs, 4 NOTs
MGA
$F = ((W + XY) \oplus ((X + Y)(X \oplus Z)))'$
7 compuertas
2 ANDs, 2 OR, 2 XORs, 1 NOT
PG Prefija
$F = (Z + (X + Y)') \oplus ((Y \oplus W) + (XY))$
7 compuertas
1 AND, 3 ORs, 2 XORs, 1 NOT

Tabla 5.6: Comparación de las mejores soluciones obtenidas por la PG Prefija, el MGA, un Diseñador Humano con Mapas de Karnaugh y la técnica de Sasao para el circuito del segundo ejemplo.

Z	W	X	Y	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Tabla 5.7: Tabla de verdad para el circuito del tercer ejemplo.

5.1.3 Ejemplo 3

El tercer ejemplo consiste en un circuito de 4 entradas y 1 salida, con la tabla de verdad que se muestra en la Tabla 5.7. Se realizaron 20 corridas con un tamaño total de población de 500 individuos y un máximo de generaciones de 1200. Este incremento en el tamaño de la población se debió al grado de complejidad de la salida del circuito. Los resultados del experimento son mostrados en la Tabla 5.8. Encontramos en el 85% de las corridas circuitos factibles y el 10% de las ocasiones fueron soluciones óptimas con 7 compuertas. La corrida 3 se encuentra cercana a la mediana y describe mejor el comportamiento de la PG Prefija en estos experimentos y la mostramos gráficamente en la Figura 5.6. La aptitud promedio de las 20 corridas nos da como resultado una media de 0.8387, con una desviación estándar de 0.0148 y la mediana en 0.8380.

La mejor solución obtenida ocurrió en la corrida 20, alcanzando una solución factible en la generación 9 con 13 compuertas y encontrando el óptimo en la generación 20 con 7 compuertas. El diagrama lógico del circuito óptimo es mostrado en la Figura 5.7. La peor solución obtenida tuvo 13 compuertas y ocurrió en la corrida 3.

No. Corrida	Aptitud	Aptitud Promedio	No. Compuertas
1	1.0000	0.8298	11
2	1.0000	0.8380	8
3	1.0000	0.8379	8
4	1.0000	0.8200	11
5	1.0000	0.8517	7
6	0.9375	0.8336	6
7	1.0000	0.8542	8
8	0.9375	0.8238	6
9	1.0000	0.8356	8
10	1.0000	0.8363	8
11	1.0000	0.8485	9
12	1.0000	0.8427	8
13	1.0000	0.8201	13
14	0.9375	0.8297	6
15	1.0000	0.8408	8
16	1.0000	0.8448	9
17	1.0000	0.8439	8
18	1.0000	0.8171	11
19	1.0000	0.8421	8
20	1.0000	0.8834	7

Tabla 5.8: Análisis de 20 corridas para el tercer ejemplo.

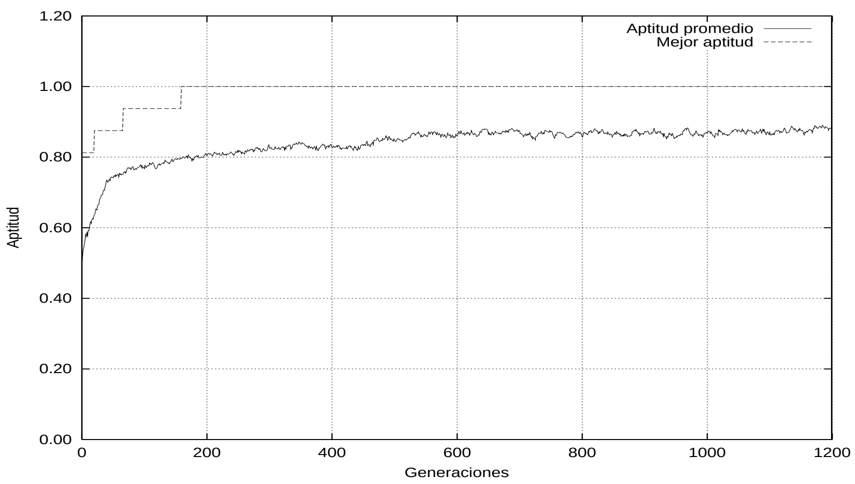

Figura 5.6: Gráfica de la corrida ubicada en la mediana del tercer ejemplo.

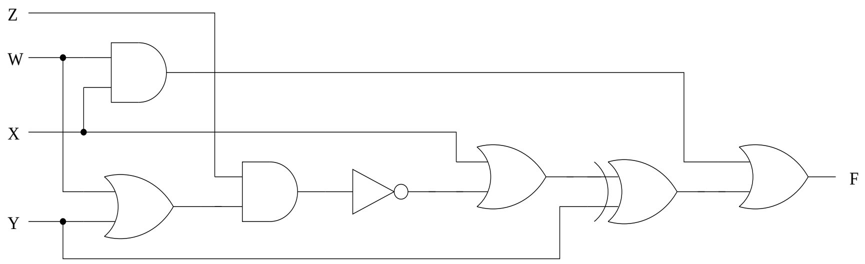

Figura 5.7: Diagrama lógico del circuito óptimo obtenido con la PG Prefija para el tercer ejemplo.

En la Tabla 5.9 se muestra un cuadro comparativo de los resultados obtenidos por otras técnicas. Encontramos que (utilizando el Método de Quine-McCluskey) el Diseñador Humano 1 desarrolla una solución con 9 compuertas, mucho mejor que la obtenida por (Mapas de Karnaugh y álgebra booleana) el Diseñador Humano 2 con 12 compuertas.

Se observó que las técnicas evolutivas logran soluciones factibles con mucha frecuencia y logran encontrar valores óptimos, en este caso, con el uso de sólo 7 compuertas. La Programación Genética Prefija utilizó 600,000 iteraciones contra las 280,000 del NGA, pero requiere un esfuerzo de cómputo mucho menor que el NGA, debido a la representación arbórea que maneja la PG Prefija.

Diseñador Humano 1
$F = (Y + ZW)' + WX + (ZX')Y$
9 compuertas
4 ANDs, 3 ORs, 2 NOTs
Diseñador Humano 2
$F = Z'Y' + W'Y' + WX + ZX'Y$
12 compuertas
5 ANDs, 3 ORs, 4 NOTs
NGA
$F = ((WX \oplus (W + Y))(X + (Z \oplus Y)))'$
7 compuertas
2 ANDs, 2 ORs, 2 XORs, 1 NOT
PG Prefija
$F = (XW) + (Y \oplus (((W + Y)Z)' + X))$
7 compuertas
2 ANDs, 3 ORs, 1 XORs, 1 NOT

Tabla 5.9: Comparación de las mejores soluciones obtenidas por la PG Prefija, el NGA y Diseñadores Humanos para el circuito del tercer ejemplo

Z	W	X	Y	F
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Tabla 5.10: Tabla de verdad para el circuito del cuarto ejemplo.

5.1.4 Ejemplo 4

El cuarto ejemplo es un circuito de 4 entradas y 1 salida. La Tabla 5.10 muestra la configuración de la tabla de verdad. Se realizaron 20 corridas con un tamaño total de población de 400 individuos y un número máximo de 1000 generaciones. Los resultados se muestran en la Tabla 5.11. Se obtuvieron un 80% de las corridas con circuitos factibles y el 20% de las ocasiones se encontraron soluciones óptimas con 6 compuertas. La corrida 13 se encuentra cercana a la mediana y describe mejor el comportamiento de dichos experimentos, como lo muestra la Figura 5.8. La aptitud promedio de las 20 corridas nos da como resultado una media de 0.8583, con una desviación estándar de 0.0082 y la mediana en 0.8601.

La mejor solución obtenida ocurrió en la corrida 12, alcanzando una solución factible en la generación 28 con 29 compuertas y encontrando el óptimo en la generación 147 con 6 compuertas. El diagrama lógico de esta solución es mostrado en la Figura 5.9. La peor solución obtenida tuvo 12 compuertas y ocurrió en la corrida 15.

No. Corrida	Aptitud	Aptitud Promedio	No. Compuertas
1	1.0000	0.8646	6
2	0.9375	0.8659	3
3	1.0000	0.8531	10
4	1.0000	0.8615	6
5	1.0000	0.8595	8
6	1.0000	0.8599	9
7	1.0000	0.8413	7
8	0.9375	0.8546	3
9	0.9375	0.8604	3
10	1.0000	0.8441	9
11	1.0000	0.8506	8
12	1.0000	0.8727	6
13	1.0000	0.8602	9
14	1.0000	0.8481	7
15	1.0000	0.8533	12
16	1.0000	0.8559	6
17	1.0000	0.8644	7
18	1.0000	0.8701	8
19	1.0000	0.8643	7
20	0.9375	0.8605	3

Tabla 5.11: Análisis de 20 corridas para el cuarto ejemplo.

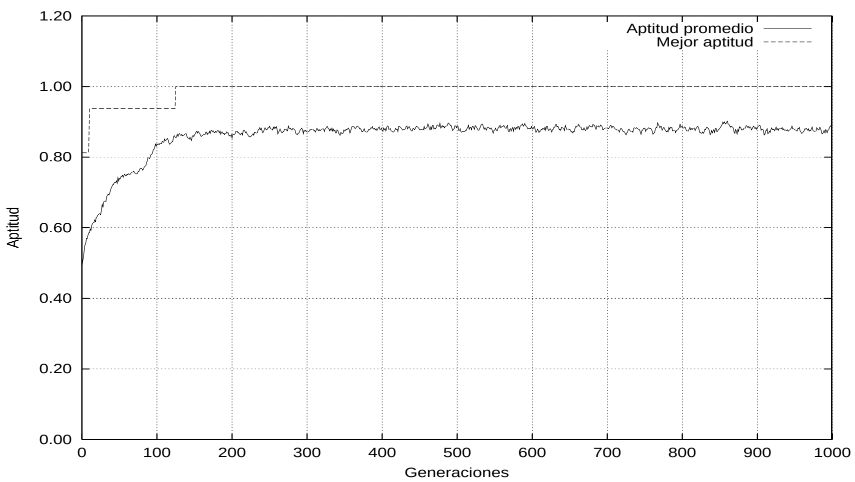

Figura 5.8: Gráfica de la corrida ubicada en la mediana del cuarto ejemplo.

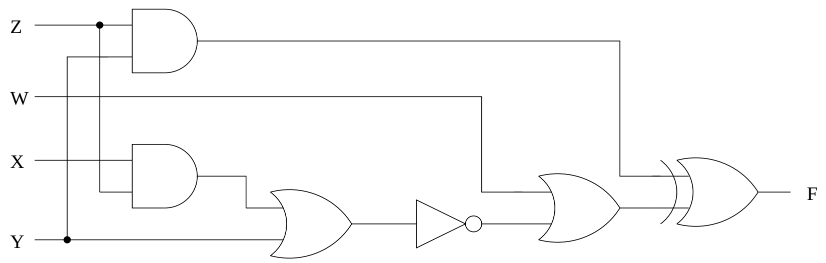

Figura 5.9: Diagrama lógico del circuito óptimo encontrado por la PG Prefija para el cuarto ejemplo.

El resultado es comparado con el Diseñador Humano que utiliza Mapas de Karnaugh y álgebra booleana, encontrando una solución de 8 compuertas (Tabla 5.12). El Sistema de Hormigas (SH) encuentra un circuito con 7 compuertas, produciendo circuitos factibles el 25% de las veces en 20 corridas aleatorias. El BGA también encontró un circuito con 7 compuertas, generando circuitos factibles el 40% de las veces en 20 corridas aleatorias. La PG Prefija encontró circuitos factibles de 7 compuertas en el 20% de las veces en 20 corridas, además del 20% logrado con circuitos de 6 compuertas.

Diseñador Humano
$F = (Y' + Z \oplus W)((ZX)' + W \oplus Y$
8 compuertas
2 ANDs, 2 ORs, 2 XORs, 2 NOTs
Sistema de Hormigas
$F = (ZX + Y)' + Z \oplus W \oplus Y'$
7 compuertas
1 AND, 2 ORs, 2 XORs, 2 NOTs
BGA
$F = (ZX + (W + Y))' + (ZY \oplus W)$
7 compuertas
2 AND, 3 ORs, 1 XOR, 1 NOT
PG Prefija
$F = (YZ) \oplus (Y + (XZ))' + W$
6 compuertas
2 ANDs, 2 ORs, 1 XORs, 1 NOTs

Tabla 5.12: Comparación de las mejores soluciones obtenidas por la PG Prefija, el BGA, el Sistema de Hormigas y un Diseñador Humano para el circuito del cuarto ejemplo.

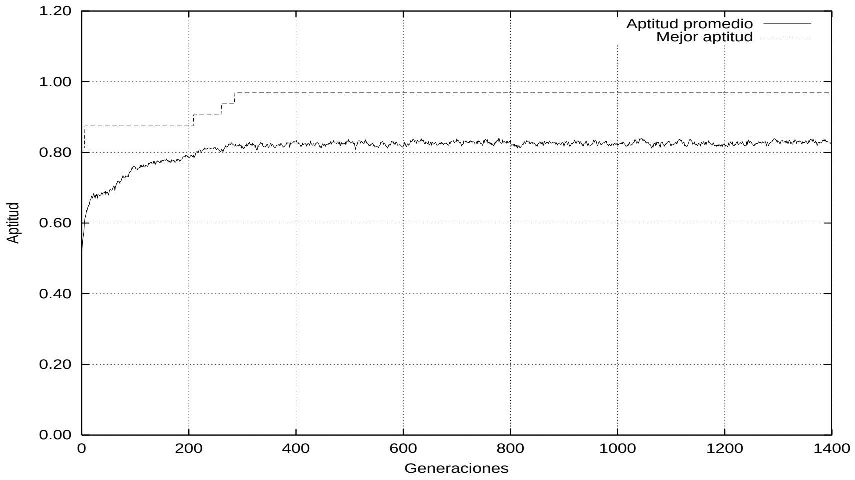

Figura 5.10: Gráfica de la corrida ubicada en la mediana del quinto ejemplo.

5.1.5 Ejemplo 5

El quinto ejemplo consiste en un circuito de 5 entradas y 1 salida. La Tabla 5.13 muestra su tabla de verdad. Se realizaron 20 corridas con un tamaño total de población de 500 individuos y un número máximo de 1400 generaciones. Los resultados son mostrados en la Tabla 5.14. El 75% de las corridas produjo circuitos factibles y el 20% de las ocasiones arrojó soluciones óptimas con 7 compuertas. La corrida 12 se encuentra cercana a la mediana y describe apropiadamente el comportamiento de PG Prefija en los experimentos realizados como lo vemos en la Figura 5.10. La aptitud promedio de las 20 corridas nos da como resultado una media de 0.8122, con una desviación estándar de 0.0178 y la mediana en 0.8088.

La mejor solución ocurrió en la corrida 19, obteniendo un circuito factible en la generación 300 con 60 compuertas y encontrando el óptimo en la generación 324 con 7 compuertas. El diagrama lógico del circuito óptimo es mostrado en la Figura 5.11. La peor solución obtenida tuvo 17 compuertas y ocurrió en la corrida 1.

V	Z	W	X	Y	F
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	1
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	1	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	0	1	1
1	1	1	1	0	0
1	1	1	1	1	1

Tabla 5.13: Tabla de verdad para el circuito del quinto ejemplo.

No. Corrida	Aptitud	Aptitud Promedio	No. Compuertas
1	1.0000	0.8067	17
2	0.9375	0.7976	9
3	1.0000	0.8257	8
4	1.0000	0.8149	8
5	0.9375	0.7923	10
6	1.0000	0.7990	10
7	1.0000	0.8286	9
8	1.0000	0.8040	13
9	1.0000	0.8454	7
10	1.0000	0.8238	7
11	1.0000	0.8294	7
12	0.9688	0.8108	16
13	1.0000	0.7995	11
14	1.0000	0.8135	8
15	1.0000	0.7951	13
16	1.0000	0.7895	14
17	1.0000	0.8240	9
18	0.9062	0.7945	8
19	1.0000	0.8516	7
20	0.9688	0.7984	12

Tabla 5.14: Análisis de 20 corridas para el quinto ejemplo.

En la Tabla 5.15 se encuentra un comparativo de los resultados obtenidos por otras técnicas para este circuito. El Diseñador Humano (DH) utiliza Mapas de Karnaugh y álgebra booleana, generando un circuito con 12 compuertas. Como observamos, el DH nunca hace uso de la compuerta XOR, que como se ha mencionado, resulta de gran utilidad al momento de simplificar. El MGA encuentra una solución con 10 compuertas utilizando una poblacion de 650 individuos y 2000 generaciones. La PG Prefija obtiene un circuito de menor tamaño usando tan solo 7 compuertas.

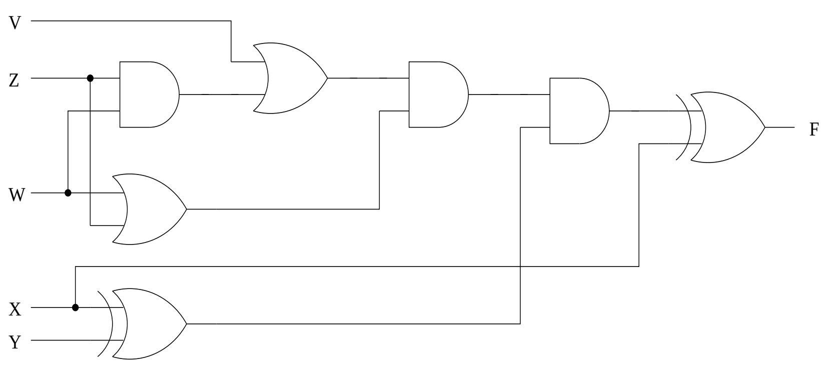

Figura 5.11: Diagrama lógico del circuito óptimo encontrado por la PG Prefija para el quinto ejemplo.

Diseñador Humano
$F = X((Z+W)' + V'(ZW)') + Y(V(Z+W) + ZW)$
12 compuertas
5 ANDs, 4 ORs, 3 NOTs
MGA
$F = ((V+W)(V+Z)(X \oplus Y)(Z+W) \oplus X')'$
10 compuertas
3 ANDs, 3 ORs, 2 XORs, 2 NOTs
PG Prefija
$F = (V+WZ)(X \oplus Y)(Z+W) \oplus X$
7 compuertas
3 ANDs, 2 ORs, 2 XORs

Tabla 5.15: Comparación de las mejores soluciones obtenidas por la PG Prefija, el MGA y un Diseñador Humano para el circuito del quinto ejemplo.

5.2 Análisis de resultados

A continuación se describirán brevemente algunos resultados obtenidos por la programación genética y se realizará una comparación con los resultados obtenidos por la PG Prefija y las otras técnicas ya mencionadas. Para efectos de dicha comparación se emplearán los teoremas booleanos vistos en la sección de álgebra booleana.

5.2.1 Ejemplo 1

Corrida 12

En la corrida 12 encontramos una solución con 4 compuertas lógicas que demuestra la factorización que puede realizarse al problema del Diseñador Humano 1 de 5 compuertas.

Aplicando la ley distributiva a la expresión del Diseñador Humano 1 obtenemos:

$$Z(X \oplus Y) + Y(X \oplus Z) = (ZX \oplus ZY) + (XY \oplus ZY)$$

Seleccionando ZY como factor común llegamos a

$$ZY \oplus (ZX + XY)$$

Aplicando nuevamente la ley distributiva en X obtenemos la misma expresión de la corrida 12 con 4 compuertas lógicas.

$$ZY \oplus (Z + Y)X$$

Finalmente se obtiene una expresión equivalente a la producida por la PG Prefija

$$\begin{array}{ccc} \text{PG Prefija} & & \text{Diseñador Humano 1} \\ ZY \oplus (Z+Y)X & = & Z(X \oplus Y) + Y(X \oplus Z) \end{array}$$

Corrida 7

La expresión de 9 compuertas obtenida en la corrida 7 es equivalente a la solución del Diseñador Humano 2, la cual quedará con 6 compuertas después de una simplificación algebraica. A continuación mostramos la expresión generada por la PG Prefija:

$$((XY \oplus Z)X)X \oplus (((Z \oplus X) + X) \oplus X)Y$$

Aplicando la ley distributiva a la expresión

$$((XY \oplus Z)X)X = (XXY \oplus ZX)X$$

Después se aplica el teorema 3 a $XX = X$

$$(XY \oplus ZX)X$$

Nuevamente aplicanmos la ley distributiva a la expresión

$$(XXY \oplus ZXX)$$

Y nuevamente aplicamos el teorema 3 a $XX = X$

$$(XY \oplus ZX)$$

Y seleccionamos X como factor común en la expresión anterior obteniendo

$$X(Y \oplus Z)$$

Ahora aplicamos la equivalencia de la compuerta XOR a la segunda parte de la expresión de la corrida 7

$$(((Z \oplus X) + X) \oplus X)Y = (X((Z \oplus X) + X)' + X'((Z \oplus X) + X))Y$$

Después aplicamos la ley de DeMorgan $(X + Y)' = X'Y'$ a la expresión $((Z \oplus X) + X)' = (Z \oplus X)'X'$ obteniendo

$$(X((Z \oplus X)'X' + X'((Z \oplus X) + X)))Y$$

Aplicando la ley distributiva

$$((Z \oplus X)'X'X + X'(Z \oplus X) + XX')Y$$

Aplicamos el teorema 4 $X'X = 0$

$$((Z \oplus X)'0 + X'(Z \oplus X) + 0)Y$$

Y aplicamos el teorema 1 $X0 = 0$

$$0 + (X'(Z \oplus X) + 0)Y$$

Después aplicamos el teorema 5 $X + 0 = X$

$$(X'(Z \oplus X))Y$$

Nuevamente aplicamos la ley distributiva

$$((ZX' \oplus XX'))Y$$

Aplicamos el teorema 4 obteniendo

$$((ZX' \oplus 0))Y$$

Y aplicamos la igualdad $X \oplus 0 = X$, también conocido como **"buffer"** de paso.

$$(ZX')Y$$

Aplicamos la ley asociativa obteniendo

$$ZX'Y$$

Unimos ambos resultados obtenidos tras la reducción algebraica

$$X(Y \oplus Z) \oplus ZX'Y$$

Finalmente obtenemos una expresión similar a la del Diseñador Humano 2 (DH2), con la diferencia de que se usa una compuerta XOR para la PG Prefija y una compuerta OR para el DH2.

$$\begin{array}{ccc} \text{PG Prefija} & & \text{Diseñador Humano 2} \\ X(Y \oplus Z) \oplus X'ZY & = & X(Y \oplus Z) + X'ZY \end{array}$$

Obtenemos el mismo resultado con cualquiera de las dos compuertas (XOR y OR) debido a la combinación de ceros y unos que produce el mismo efecto con OR y con XOR como se muestra a continuación con la evaluación de ambos lados de las expresiones:

$$\begin{array}{lcl} X(Y \oplus Z) & = & 00000110 \\ X'ZY & = & 00010000 \end{array}$$

Corrida 16

A partir del resultado obtenido de la corrida 16 con una expresión que cuenta con 6 compuertas lógicas, se elabora una reducción algebraica que nos conduce a una solución de 4 compuertas lógicas igual a la solución obtenida por el MGA. A continuación se presenta la expresión de la corrida 16.

$$YY(Z \oplus X) + (Y \oplus Z)X$$

Aplicamos el teorema 3: $YY = Y$ obteniendo

$$Y(Z \oplus X) + (Y \oplus Z)X$$

Aplicamos la ley distributiva

$$(YZ \oplus YX) + (YX \oplus ZX)$$

Seleccionamos YX como factor común

$$YX \oplus (YZ + ZX)$$

Después seleccionamos Z como factor común

$$YX \oplus (Y + X)Z$$

Finalmente obtenemos una solución de 4 compuertas lógicas similar a la lograda por el MGA.

$$\begin{array}{ccc} \text{PG Prefija} & & \text{MGA} \\ YX \oplus (Y + X)Z & = & YX \oplus (Y + X)Z \end{array}$$

Corridas 2 y 3

Los resultados obtenidos por la PG Prefija varían en complejidad, aún para las soluciones óptimas, en donde se cuenta con diferentes expresiones que tienen una distinta distribución de los términos y las funciones. Estas expresiones producen todas la salida válida para el circuito, como es el caso de las soluciones de las corridas 2 y 3 respectivamente. Puede verse fácilmente que ambas soluciones son equivalentes:

$$(YZ \oplus X)(Z + Y) = (YX \oplus Z)(X + Y)$$

Corrida 9

A continuación describiremos el proceso de evolución en la corrida 9 una vez encontrado un valor factible. Se observa cómo el proceso de elitismo y la recombinación de código generan expresiones más cortas hasta convertirla en una solución factible con menos componentes.

En la generación 24 se encuentra la solución factible con 6 compuertas como se muestra a continuación:

&|X^|XXY^&YXZ

En la generación 28 cambia la subcadena |XX por &XX sin mayor efecto en el resultado de la expresión

&|X^&XXY^&YXZ

En la generación 29 reemplaza la subcadena &XX por el término X

&|X^XY^&YXZ

En la generación 82 reemplaza la subcadena ^XY por el término Y, sin afectar el resultado en la expresión final

&|XY^&YXZ

El resultado es una expresión de 4 compuertas.

$$F = (X + Y)((YX) \oplus Z)$$

5.2.2 Ejemplo 2

Corrida 16

En la corrida 16 se obtiene un circuito que podría ser comparado con el obtenido por Sasao de 12 compuertas ya que es el único que se compone únicamente de compuertas XOR, AND y NOT. La única diferencia es que el circuito obtenido por la PG Prefija contiene 10 compuertas (en vez de 12). Esto se debe a que la PG Prefija hace una reutilización de la combinación $Z \oplus W$, como se muestra a continuación.

$$\begin{gathered}\text{PG Prefija}\\ (Z \oplus W) \oplus Y \oplus (X \oplus Y)'(Z \oplus W)X' \oplus Z \oplus X\\ =\\ \text{Sasao}\\ X' \oplus Y'W' \oplus XY'Z' \oplus X'Y'W\end{gathered}$$

Sin embargo, no se pudo encontrar una equivalencia lógica al circuito de Sasao que nos permitiera realizar alguna reducción a esta solución.

Corridas 2 y 7

Los mejores resultados factibles varían en complejidad y en componentes al usar la PG Prefija. De esta manera se desarrollan expresiones muy variadas como las producidas en la corrida 2 y 7 respectivamente, que son mostradas a continuación:

$$Z(Y + X) \oplus (XY + W) \oplus X' = ((X + Y)' + Z) \oplus (XY + (\oplus W))'$$

Corrida 19

El proceso evolutivo que se desarrolla para encontrar la mejor solución factible en la corrida 19 es descrito a continuación. Podemos observar cómo de un circuito factible de 26 compuertas en la generación 103, se llega a uno de sólo 7 compuertas en la generación 195, tal y como se muestra a continuación:

^!!^&XX|W&|&|Y!!|XX&^&&!|ZXXZXY!XX!&|Y!!|XXZ

En la generación 106, se reemplaza la subcadena &&!|ZXXZ por &!XZ sin afectar el resultado, obteniendo

^!!^&XX|W&|&|Y!!|XX&^&!XZXY!XX!&|Y!!|XXZ

En la generación 112 se recombinan varios fragmentos de la cadena generando

^!!^&XX|W&&Y!&&!ZY^!WXX!&|Y!!|XXZ

En la generación 113 se reemplaza la subcadena |XX por el término X

^!!^&XX|W&&Y!&&!ZY^!WXX!&|Y!!XZ

En la generación 114 se cambia !W por X generando la expresión

^!!^&XX|W&&Y!&&!ZY^XXX!&|Y!!XZ

En la generación 115 se reemplaza &&!ZY^XX por el término W sin afectar el resultado, obteniendo la expresión

^!!^&XX|W&&Y!WX!&|Y!!XZ

En la generación 122 se reemplaza la subcadena &XX por el término X

^!!^X|W&&Y!WX!&|Y!!XZ

En la generación 129 se cambia !W por Y

^!!^X|W&&YYX!&|Y!!XZ

En la generación 139 se reemplaza la subcadena !!X por el término X

^!!^X|W&&YYX!&|YXZ

En la generación 165 se cambia la subcadena &YY por el término Y

^!!^X|W&YX!&|YXZ

En la generación 195 se elimina la doble negación y se reduce la expresión a 7 compuertas lógicas

^^X|W&YX!&|YXZ

El resultado obtenido es muy *similar* al producido por el MGA. Podemos observar los componentes $YX + W$ y $Y + X$ en ambos resultados.

$$\begin{array}{c} \text{PG Prefija} \\ (YX + W) \oplus X \oplus ((Y + X)Z)' \\ = \\ \text{MGA} \\ ((W + XY) \oplus ((X + Y)(X \oplus Z)))' \end{array}$$

5.2.3 Ejemplo 3

Corrida 15

El circuito generado en la corrida 15 cuenta con 8 compuertas lógicas y es muy similar al producido por el Diseñador Humano 1 (DH1) de 9 compuertas. Ambos circuitos sólo varían en un componente.

$(ZX')Y$ del Diseñador Humano 1 y $(X \oplus Y)Z$ en PG Prefija.

Sin embargo, no son equivalentes, pero la configuración de salida no genera cambios en el resultado de la expresión.

$$\begin{array}{ccc} \text{PG Prefija} & & \text{Diseñador Humano 1} \\ (Y + ZW)' + WX + (X \oplus Y)Z & = & (Y + ZW)' + WX + (ZX')Y \end{array}$$

Corrida 5

El resultado obtenido por el NGA es muy similar en componentes a uno obtenido en la corrida 5, ambos con 7 compuertas. Existen en ambas expresiones componentes similares, pero no existe alguna equivalencia lógica.

$$\begin{array}{c} \text{PG Prefija} \\ WX + ((Y + W)((Z \oplus Y) + X))' \\ = \\ \text{NGA} \\ ((WX \oplus (W + Y))(X + (Z \oplus Y)))' \end{array}$$

Corridas 5 y 20

Los mejores resultados factibles producidos por la PG Prefija pueden ser muy similares en componentes pero su distribución difiere bastante de una expresión a otra como se muestra a continuación con los resultados de las corridas 5 y 20 respectivamente con 7 compuertas cada una:

$$WX + ((Y + W)((Z \oplus Y) + X))' = XW + (((W + Y)Z)' + X) \oplus Y$$

Corrida 20

El proceso evolutivo generado para producir la mejor solución factible de la corrida 20 es descrito a continuación. Se genera la expresión factible con 13 compuertas lógicas en la generación 9 como se muestra a continuación.

|&X&||&|!XWZXWW^Y|!&|WYZX

En la generación 10 se reemplaza la subcadena ||&|!XWZXW por |WY sin afectar el resultado de la expresión

|&X&|WYW^Y|!&|WYZX

En la generación 14 se reemplaza la subcadena |WY por el término X

|&X&XW^Y|!&|WYZX

En la generación 16 se reemplaza la subcadena &X&XW por el término &XW generando la expresión

|&XW^Y|!&|WYZX

El resultado es una expresión lógica de 7 compuertas lógicas.

$$F = XW + (((W + Y)Z)' + X) \oplus Y$$

5.2.4 Ejemplo 4

Corrida 16

En la corrida 16 encontramos un circuito factible de 6 compuertas que presenta los mismos componentes del producido por el Sistema de Hormigas (SH) el cual cuenta con 7 compuertas. Después de un análisis se encontró una reducción algebraica de la expresión, lo que nos conduce a ahorrar una compuerta como se muestra a continuación en la expresión del SH:

$$(ZX+Y)' + (Z \oplus W) \oplus Y'$$

Después de la comparación de resultados obtenemos una nueva equivalencia lógica que puede resultar de utilidad en la reducción algebraica:

$$A' + (B \oplus C') = (A(B \oplus C))'$$

$$\text{aplicando leyes de DeMorgan } (A(B \oplus C))' = A' + (B \oplus C)'$$

$$\text{encontramos que } (B \oplus C') \equiv (B \oplus C)'$$

Donde

$$A = (ZX+Y) \quad B = (Z \oplus W) \quad C = Y$$

$$((ZX+Y)(Z \oplus W \oplus Y))'$$

Esto da como resultado la misma expresión que la generada en la corrida 16, la cual tiene 6 compuertas lógicas.

PG Prefija		Sistema de Hormigas
$((ZX+Y)(Z \oplus W \oplus Y))'$	$=$	$(ZX+Y)' + (Z \oplus W) \oplus Y'$

Corrida 17

La expresión del BGA con 7 compuertas es muy similar en componentes a la producida por la corrida 17, la cual tiene igual número de compuertas. Sin embargo, las expresiones $((Z \oplus W)X \oplus Y)'$ y $(ZX + (W+Y))'$ no son equivalentes:

PG Prefija		BGA
$((Z \oplus W)X \oplus Y)' + (ZY \oplus W)$	$=$	$(ZX + (W+Y))' + (ZY \oplus W)$

Corridas 4 y 12

En las corridas 4 y 12 se generan circuitos factibles de 6 compuertas. Ambas soluciones son equivalentes. Sin embargo, difieren en el número de compuertas XOR y OR. Este efecto es debido a la recombinación de código que permite insertar subcadenas aptas que no afectan el desempeño final de la expresión:

Corrida 4		Corrida 12
$((W \oplus Y \oplus Z)(XZ+Y))'$	$=$	$YZ \oplus ((XZ+Y)' + W)$
2 ANDs 2 XORs 1 ORs 1 NOTs		2 ANDs 1 XORs 2 ORs 1 NOTs

Corrida 1

En la primer corrida efectuada para este ejemplo se obtiene una solución factible en la generación 178 con 21 compuertas lógicas. El proceso evolutivo hasta llegar a una mejor solución factible debido al elitismo es descrito a continuación:

!&&^^YZW|^|^Y&Z^WW^YX&^&^^YXWYW^WW^YX||ZZY

En la generación 182 se reemplaza la subcadena &^^YXWY por ^YX

!&&^^YZW|^|^Y&Z^WW^YX&^^YXW^WW^YX||ZZY

En la generación 187 se reemplaza la subcadena ^Y&Z^WW por el término X y el término X por Y, generando la expresión

!&&^^YZW|^|X^YX&^^YXW^WW^YY||ZZY

En la generación 189 se cambia la subcadena |^|X^YX&^^YXW^WW ^YY por |Y^YX

!&&^^YZW|Y^YX||ZZY

En la generación 203 se reemplaza la subcadena ||Z por el término Z generando

!&&^^YZW|Y^YX|ZY

En la generación 208 se reemplaza la subcadena &^^YZW|Y^YX|ZY por ^^ZYW|&ZXY produciéndose la expresión

!&^^ZYW|&ZXY

El resultado es una expresión de 6 compuertas lógicas:

$$F = ((ZX + Y)(W \oplus Y \oplus Z)'$$

5.2.5 Ejemplo 5

Corrida 14

Se encontró un circuito de 8 compuertas en la corrida 14, el cual es muy similar al producido por el Diseñador Humano con 12 compuertas. Ambas son equivalentes, pero la reutilización de compuertas de la corrida 14 hace posible el uso de sólo 8 compuertas lógicas.

$$\begin{array}{c} \text{PG Prefija} \\ (WV+Z)(V+W)Y+((WV+Z)(V+W)X)\oplus X \\ = \\ \text{Diseñador Humano} \\ X((Z+W)'+V'(ZW)')+Y(V(Z+W)+ZW) \end{array}$$

Corrida 19

El resultado de la corrida 19 con 7 compuertas nos permite generar una nueva equivalencia que explica la reducción que podría generarse con el resultado obtenido por el MGA de 10 compuertas lógicas. A continuación se muestra la expresión generada por el MGA:

$$((V+W)(V+Z)(X\oplus Y)(Z+W)\oplus X')'$$

Si aplicamos la ley distributiva a la expresión $(V+W)(V+Z) = V+WZ$, obtenemos la expresión

$$((V+WZ)(X\oplus Y)(Z+W)\oplus X')'$$

Tras el análisis de la expresión del MGA simplificada y de la solución de la corrida 19 encontramos la siguiente equivalencia lógica

$$(ABC\oplus D')' = ABC\oplus D$$

$$\text{sabemos que } (B\oplus C') \equiv (B\oplus C)'$$

$$(ABC\oplus D')' = (ABC\oplus D)'' = ABC\oplus D$$

Donde

$$\begin{array}{ll} A=(Z+W) & B=(V+WZ) \\ C=(X\oplus Y) & D=X \end{array}$$

$$(V+WZ)(X\oplus Y)(Z+W)\oplus X$$

Esto da como resultado la misma expresión que la generada en la corrida 19, la cual tiene 7 compuertas lógicas.

$$\begin{gathered}\text{PG Prefija}\\ (V+WZ)(X\oplus Y)(Z+W)\oplus X\\ =\\ \text{MGA}\\ ((V+W)(V+Z)(X\oplus Y)(Z+W)\oplus X')'\end{gathered}$$

Corrida 17

La expresión resultante de la corrida 17 tiene 9 compuertas lógicas, pero puede ser reducida a 7 compuertas lógicas. Tras un análisis se encontró una nueva equivalencia lógica que puede resultar de utilidad en el álgebra booleana. A continuación mostramos la expresión generada en la corrida 17:

$$X\oplus((W+Z)(W+V)(V+Z))X+((W+Z)(W+V)(V+Z))Y$$

Aplicando la ley distributiva a la expresión $(W+V)(V+Z)=(V+WZ)$, obtenemos la expresión

$$X\oplus((W+Z)(V+WZ))X+((W+Z)(V+WZ))Y$$

Analizando esta expresión con la generada en la corrida 19 obtenemos una nueva equivalencia que se muestra a continuación:

$$(A\oplus B)A+CB=A\oplus B(A\oplus C)$$

Donde

$$\begin{aligned}A&=X\\ B&=((W+Z)(V+WZ))\\ C&=Y\end{aligned}$$

$$X\oplus(W+Z)(V+WZ)(X\oplus Y)$$

Esto da como resultado la misma expresión generada en la corrida 19 con 7 compuertas.

Corridas 9 y 11

Los resultados obtenidos por la PG Prefija varían en complejidad, configuración y hasta número de compuertas necesarias para diseñar un circuito factible. Los casos de las corridas 9 y 11 demuestran la capacidad de la PG Prefija para el diseño lógico.

Corrida 9		Corrida 11
$X \oplus (X \oplus Y)(WZ + (W \oplus Z)V))$	$=$	$X \oplus (V + Z)(W + VZ)(X \oplus Y)$
3 ANDs 3 XORs 1 ORs		3 ANDs 2 XORs 2 ORs
7 compuertas		7 compuertas

Corrida 10

En la corrida 10 se encuentra una solución factible en la generación 637 con 12 compuertas lógicas y es reducida a 7 compuertas por el proceso de elitismo. El proceso de reducción de componentes es mostrado a continuación:

^&|^&W!^XV&VZW&||VZZ^XYX

En la generación 651 se reemplaza la subcadena |VZ por el término V

^&|^&W!^XV&VZW&|VZ^XYX

En la generación 658 se reemplaza la subcadena !^XV por !V

^&|^&W!V&VZW&|VZ^XYX

En la generación 698 se reemplaza la subcadena !V por el término X

^&|^&WX&VZW&|VZ^XYX

En la generación 706 se reemplaza la subcadena &WX por el término W

^&|^W&VZW&|VZ^XYX

El resultado es una expresión con 7 compuertas lógicas.

$$F = X \oplus (W + VZ)(V + Z)(X \oplus Y)$$

Capítulo 6

Circuitos de múltiples salidas

6.1 El problema de múltiples salidas

El proceso de diseñar circuitos de múltiples salidas resulta más complicado para los diseñadores humanos, ya que requieren hacer el diseño por separado para cada una de las salidas y luego se busca (mediante inspección visual) lograr una reutilización de las compuertas lógicas. Las técnicas evolutivas como el BGA [11, 13], el NGA [12, 17, 16], el MGA [19, 18] y el Sistema de Hormigas [20, 21], utilizan una representación matricial para solucionar el circuito requerido, tratando de maximizar la reutilización de compuertas. La PG Prefija requiere modificaciones adicionales en la estructura de los individuos para poder solucionar problemas de múltiples salidas.

En lugar de utilizar una expresión prefija única, los individuos constan de varias expresiones prefijas. Es decir, cada individuo se compone de un bosque de árboles. Dentro del bosque, cada árbol o expresión prefija constituye una de las salidas del circuito (Figura 6.1).

Cada árbol dentro del individuo, se especializa en su propia solución y coevoluciona en paralelo con el resto de los árboles [47]. Adicionalmente, los operadores genéticos ya implementados anteriormente tuvieron que ser modificados (este es el caso de la cruza y la mutación). También se hubo de agregar algunos operadores asexuales para asegurar la compartición de código entre los árboles de un mismo individuo.

Los operadores de cruza y mutación son básicamente los mismos, con la variante de que además de seleccionar un individuo, se tiene que seleccionar un árbol de los m que lo componen. En este punto se implementó un proceso adicional para poder detectar cúal de las salidas del circuito requiere un mayor esfuerzo. Es decir, la salida del circuito con un valor de aptitud

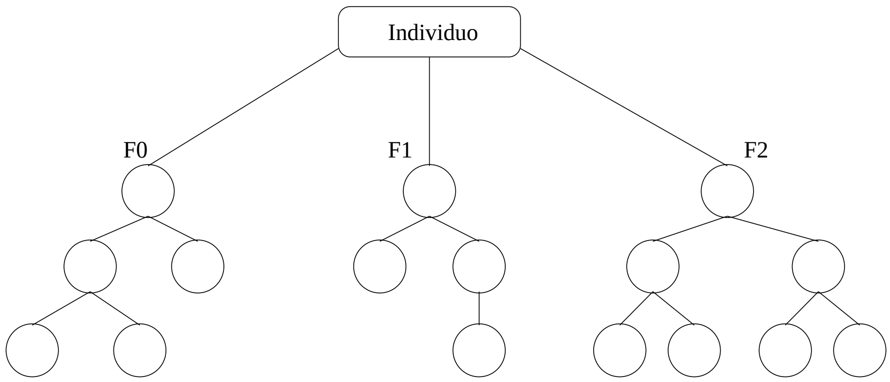

Figura 6.1: Estructura de un individuo para el problema de múltiples salidas.

menor tendrá que tener un sesgo adicional para poder ser seleccionada con mayor frecuencia y estabilizar su proceso de evolución con respecto a las demás salidas. Esta tecnica de aprendisaje por reforzamiento es conocida como Greedy [63].

La **Permutación** consiste en seleccionar de 2 árboles, 2 subárboles válidos e intercambiarlos de un árbol al otro dentro del mismo individuo en forma aleatoria, intentando maximizar las ocurrencias de código similar.

El operador **CopyPaste**, consiste en duplicar e insertar un subárbol o árbol factible en algún punto aleatorio dentro de otro árbol. Al darle preferencia a expresiones factibles, intentamos maximizar la compartición de código "útil". Esta operación, al igual que la permutación, es asexual.

En todos los operadores genéticos se implementó una selección de nodos tipo función sobre nodos tipo término [2], ya que una función nos permite manipular un mayor número de elementos. Esto equivale a que se transfiriera un número mayor de módulos complejos que de términos sencillos. Este sesgo fue del 70% a nodos tipo función y del 30% a nodos tipo término.

Finalmente, el proceso de elitismo sufre modificaciones, debido a que contamos con m salidas para un circuito. Esto implica que no siempre el mejor individuo global será la opción mas viable de transferencia genética. Como un agregado del anterior se construye un vector de elitismo que contiene un porcentaje de los mejores árboles de cada salida del circuito, permitiendo y preservando las buenas soluciones. Por último se construye un super individuo (solución ideal), que se compone de cada una de las mejores salidas y es transferido a la siguiente generación con la seguridad de que éste será el mejor individuo.

6.2 Experimentos

Los siguientes circuitos están compuestos de varias entradas y varias salidas. Se seleccionaron 3 circuitos lógicos de múltiples salidas, de distinto grado de complejidad, para ser probados con la PG Prefija. Los resultados serán comparados con las mismas técnicas expuestas en el capítulo de circuitos de una salida.

Para cada circuito se efectuaron 20 corridas en forma aleatoria, con el objetivo de realizar un análisis y medir la efectividad de la técnica generando circuitos factibles. El número de generaciones y el tamaño de la población varía según la complejidad del circuito. Todos los individuos constan de una profundidad máxima de 128 nodos. Después de varias pruebas se decidió utilizar un porcentaje de cruza del 95%, un porcentaje de mutación del 5%, un porcentaje de permutación del 5% y un porcentaje de copypaste del 5%. En todos los casos se utilizó un porcentaje del 1% para el vector de elitismo y se recomienda no exceder del 3% ya que esto provoca un declive en la diversidad.

6.2.1 Ejemplo 1

El primer ejemplo consiste en un sumador de 2 bits que consta de 4 entradas y 3 salidas, con la tabla de verdad que se muestra en la Tabla 6.1. Se realizaron 20 corridas con un tamaño total de población de 500 individuos y un número máximo de 800 generaciones. Los resultados obtenidos son mostrados en la Tabla 6.2. El 80% de las corridas produjo circuitos factibles y el 20% de las ocasiones arrojó soluciones óptimas con 7 compuertas. La corrida 19 se encuentra cercana a la mediana y describe mejor el comportamiento de la PG Prefija. Su desempeño es mostrado en la Figura 6.2. La aptitud promedio de las 20 corridas nos da como resultado una media de 0.8309, con una desviación estándar de 0.0135 y la mediana en 0.8335.

La peor solución obtenida alcanzó 21 compuertas en la corrida 18 y la mejor solución ocurrió en la corrida 12, alcanzando una solución factible en la generación 19 con 25 compuertas. El óptimo (con 7 compuertas) fue hallado en la generación 45. El diagrama lógico del circuito óptimo es mostrado en la Figura 6.3.

El resultado es comparado en la Tabla 6.3. El Diseñador Humano, utilizando mapas de Karnaugh y álgebra booleana para simplificar el circuito, generando una solución con 12 compuertas. El Algoritmo Genetico Binario logra, al igual que la Programación Genética Prefija, una solución con 7 compuertas.

Z	W	X	Y	F0	F1	F2
0	0	0	0	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	1	0
0	0	1	1	1	1	0
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	1	1	0
0	1	1	1	0	0	1
1	0	0	0	0	1	0
1	0	0	1	1	1	0
1	0	1	0	0	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	0	0	1
1	1	1	0	1	0	1
1	1	1	1	0	1	1

Tabla 6.1: Tabla de verdad para el circuito del primer ejemplo: un sumador de 2 bits.

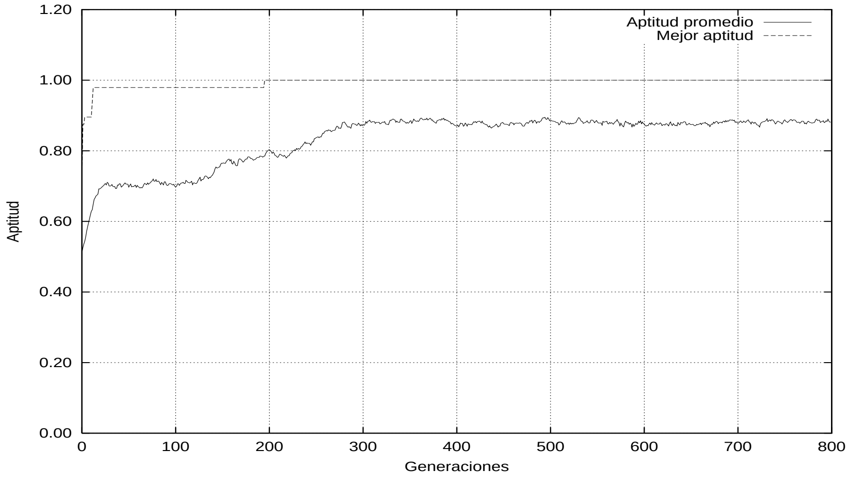

Figura 6.2: Gráfica de la corrida ubicada en la mediana del primer ejemplo: un sumador de 2 bits.

No. Corrida	Aptitud	Aptitud Promedio	No. Compuertas
1	1.0000	0.8149	8
2	0.9792	0.8164	7
3	1.0000	0.8365	11
4	1.0000	0.8334	8
5	1.0000	0.8155	11
6	1.0000	0.8211	7
7	1.0000	0.8184	7
8	1.0000	0.8501	8
9	1.0000	0.8129	7
10	0.9792	0.8101	7
11	1.0000	0.8307	10
12	1.0000	0.8363	7
13	0.9792	0.8257	7
14	1.0000	0.8372	14
15	1.0000	0.8547	9
16	1.0000	0.8492	9
17	0.9792	0.8438	7
18	1.0000	0.8428	21
19	1.0000	0.8335	8
20	1.0000	0.8339	14

Tabla 6.2: Análisis de 20 corridas para el primer ejemplo: un sumador de 2 bits.

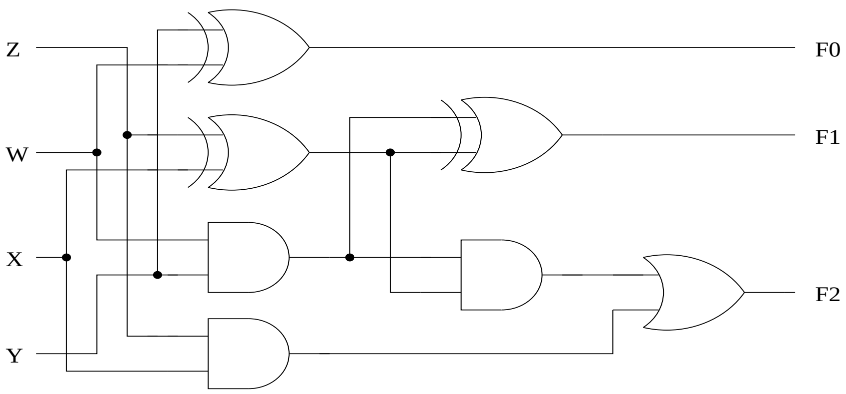

Figura 6.3: Diagrama lógico del mejor circuito obtenido por la PG Prefija para el primer ejemplo: un sumador de 2 bits.

Observamos que las salidas de ambos circuitos son exactamente las mismas. Ambas técnicas evolutivas utilizaron el mismo número de iteraciones (400,000) para encontrar la solución óptima.

Diseñador Humano
$F0 = W \oplus Y$ $F1 = (Z \oplus X)Y' + ((Z \oplus X) \oplus W)Y$ $F2 = ZX + WY(Z + X)$
12 compuertas
5 ANDs, 3 ORs, 3 XORs, 1 NOT
BGA
$F0 = W \oplus Y$ $F1 = (Z \oplus X) \oplus WY$ $F2 = ZX + WY(Z \oplus X)$
7 compuertas
3 ANDs, 1 OR, 3 XORs
PG Prefija
$F0 = Y \oplus W$ $F1 = YW \oplus (X \oplus Z)$ $F2 = YW(X \oplus Z) + XZ$
7 compuertas
3 ANDs, 1 OR, 3 XORs

Tabla 6.3: Comparación de las mejores soluciones obtenidas por un Algoritmo Genético Binario (BGA), Programación Genética Prefija (PG Prefija) y mapas de Karnaugh para el primer ejemplo: un sumador de 2 bits.

Z	W	X	Y	F0	F1	F2	F3
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	1	0	0	0
0	1	1	0	0	1	0	0
0	1	1	1	1	1	0	0
1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0
1	0	1	0	0	0	1	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	1	1	0	0
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Tabla 6.4: Tabla de verdad para el circuito del segundo ejemplo: el multiplicador de 2 bits.

6.2.2 Ejemplo 2

El siguiente ejemplo consiste en un multiplicador de 2 bits que consta de 4 entradas y 4 salidas, con la tabla de verdad que se muestra en la Tabla 6.4. Se realizaron 20 corridas con un tamaño total de población de 300 individuos y un máximo de 400 generaciones. Los resultados del experimento son mostrados en la Tabla 6.5. Encontramos en el 100% de las corridas circuitos factibles y el 15% de las ocasiones se produjeron soluciones óptimas con 7 compuertas. La corrida 8 se encuentra cercana a la mediana y describe mejor el comportamiento de los experimentos como lo vemos en la Figura 6.4. La aptitud promedio de las 20 corridas nos da como resultado una media de 0.7741, con una desviación estándar de 0.0136 y la mediana en 0.7741.

La mejor solución obtenida fue en la corrida 6, alcanzando una solución factible en la generación 38 con 14 compuertas y encontrando el óptimo de 7 compuertas en la generación 280. El diagrama lógico del circuito óptimo es mostrado en la Figura 6.5. La peor solución obtenida alcanzó 11 compuertas en la corrida 7.

No. Corrida	Aptitud	Aptitud Promedio	No. Compuertas
1	1.0000	0.8009	8
2	1.0000	0.7770	8
3	1.0000	0.7683	8
4	1.0000	0.7694	8
5	1.0000	0.7697	7
6	1.0000	0.7682	7
7	1.0000	0.7506	11
8	1.0000	0.7751	8
9	1.0000	0.7730	8
10	1.0000	0.7834	8
11	1.0000	0.7950	8
12	1.0000	0.7724	8
13	1.0000	0.7867	8
14	1.0000	0.7584	11
15	1.0000	0.7662	8
16	1.0000	0.7920	7
17	1.0000	0.7814	8
18	1.0000	0.7830	8
19	1.0000	0.8004	9
20	1.0000	0.7623	8

Tabla 6.5: Análisis de 20 corridas para el segundo ejemplo: el multiplicador de 2 bits

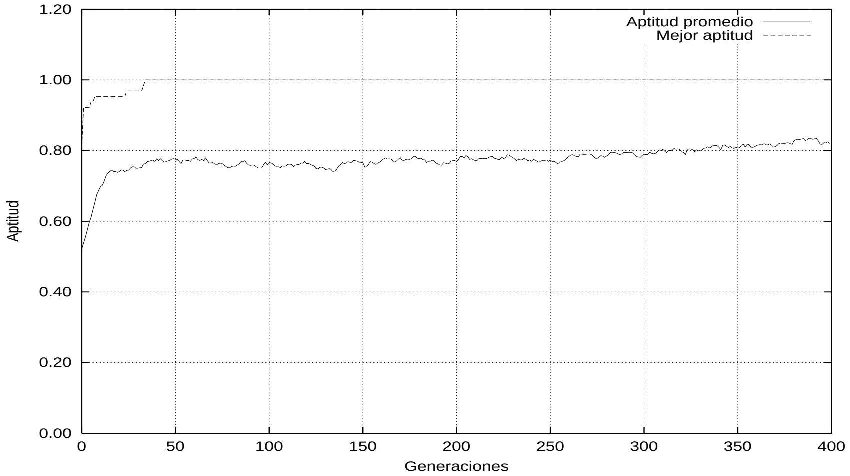

Figura 6.4: Gráfica de la corrida ubicada en la mediana del segundo ejemplo: el multiplicador de 2 bits.

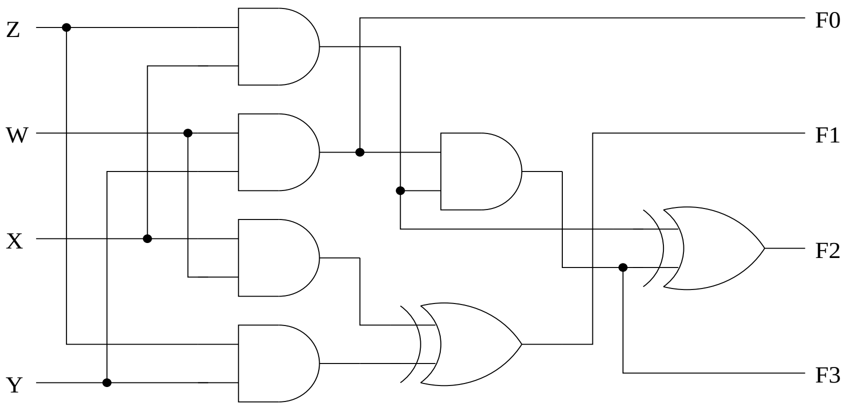

Figura 6.5: Diagrama lógico del mejor circuito producido por la PG Prefija para el segundo ejemplo: el multiplicador de 2 bits.

Diseñador Humano	Miller et al.
$F0 = WY$ $F1 = WX \oplus ZY$ $F2 = ZX(WY)'$ $F3 = WYZX$	$F0 = WY$ $F1 = ZY \oplus WX$ $F2 = (WY)'ZX$ $F3 = (ZY \oplus WX)'(ZY)$
8 compuertas	9 compuertas
6 ANDs, 1 XOR, 1 NOT	6 ANDs, 1 XOR, 2 NOTs
PG Prefija	MGA
$F0 = WY$ $F1 = XW \oplus ZY$ $F2 = (WYXZ) \oplus XZ$ $F3 = WYXZ$	$F0 = WY$ $F1 = WX \oplus ZY$ $F2 = ZX \oplus (WYZX)$ $F3 = WYZX$
7 compuertas	7 compuertas
5 ANDs, 2 XORs	5 ANDs, 2 XORs

Tabla 6.6: Comparación de las mejores soluciones obtenidas por la Programación Genética Prefija, el Algoritmo Genético Multiobjetivo, Miller et al. y un Diseñador Humano para el segundo ejemplo: el multiplicador de 2 bits.

En la Tabla 6.6 se muestra un cuadro comparativo de los resultados obtenidos por otras técnicas. Encontramos que Miller et al. [54] soluciona el circuito con 9 compuertas después de casi 3,000,000 evaluaciones. El Diseñador Humano, encuentra una solución de 8 compuertas. El MGA encuentra una solución óptima con 7 compuertas después de 325,000 evaluaciones. La PG Prefija utilizó 120,000 iteraciones para encontrar la solución de 7 compuertas que es igual a la obtenida por el MGA.

Z	W	X	Y	F0	F1	F2
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
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 6.7: Tabla de verdad para el circuito del tercer ejemplo: un comparador con 4 entradas 3 salidas.

6.2.3 Ejemplo 3

El tercer ejemplo consiste en un circuito de 4 entradas y 3 salidas, con la tabla de verdad que se muestra en la Tabla 6.7. Se realizaron 20 corridas con un tamaño total de población de 700 individuos y un número máximo de 2000 generaciones. Cabe mencionar que este circuito es de una gran complejidad y no resulta nada fácil su **"optimización"**. Los resultados del experimento son mostrados en la Tabla 6.8. Encontramos en el 60% de las corridas circuitos factibles y sólo en una ocasión se encontró una solución de 15 compuertas. La corrida 1 se encuentra cercana a la mediana y describe mejor el comportamiento de la PG Prefija como lo vemos en la Figura 6.6. La aptitud promedio de las 20 corridas nos da como resultado una media de 0.8226, con una desviación estándar de 0.0117 y la mediana en 0.8250.

La mejor solución obtenida ocurrió en la corrida 12, alcanzando una solución factible en la generación 247 con 146 compuertas y encontrando una solución con 15 compuertas en la generación 586. El diagrama lógico del circuito de 15 compuertas es mostrado en la Figura 6.7. La peor solución obtenida llegó a 30 compuertas en la corrida 6.

No. Corrida	Aptitud	Aptitud Promedio	No. Compuertas
1	1.0000	0.8258	19
2	1.0000	0.8318	16
3	0.9792	0.8283	20
4	1.0000	0.8313	20
5	0.9792	0.8311	21
6	1.0000	0.8349	30
7	1.0000	0.8368	18
8	0.9792	0.8166	19
9	0.9583	0.8362	16
10	0.9792	0.8133	19
11	1.0000	0.8232	19
12	1.0000	0.7987	15
13	0.9792	0.8141	21
14	1.0000	0.8242	26
15	0.9792	0.8093	20
16	1.0000	0.8025	20
17	1.0000	0.8067	21
18	1.0000	0.8323	18
19	1.0000	0.8220	19
20	1.0000	0.8319	19

Tabla 6.8: Análisis de 20 corridas para el tercer ejemplo: un comparador con 4 entradas 3 salidas.

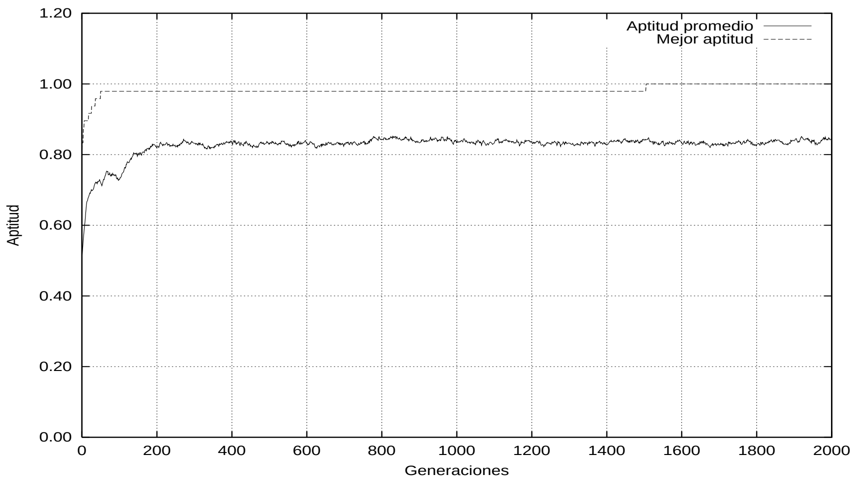

Figura 6.6: Gráfica de la corrida ubicada en la mediana del tercer ejemplo: un comparador con 4 entradas 3 salidas.

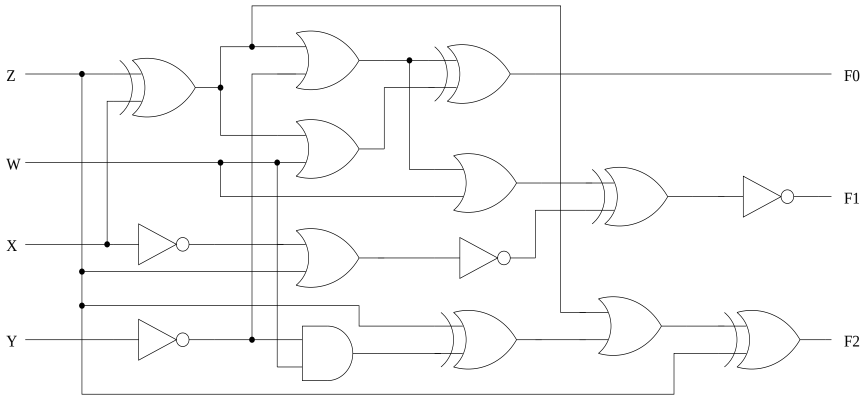

Figura 6.7: Diagrama lógico del mejor circuito obtenido por la PG Prefija para el tercer ejemplo: un comparador con 4 entradas y 3 salidas.

El resultado es comparado con el Diseñador Humano 1 que utiliza mapas de Karnaugh y álgebra booleana, obteniendo 19 compuertas (Tabla 6.9). El Diseñador Humano 2 utiliza el método de Quine-McCluskey obteniendo 13 compuertas y es claro el nivel de eficiencia que logra el MGA encontrando una solución de 9 compuertas con un alto grado de reutilización de componentes del circuito. La PG Prefija sólo logra llegar a una solución de 15 compuertas, con un grado muy bajo de reutilización de compuertas.

Diseñador Humano 1
$F0 = (Z \oplus X)'(W \oplus Y)'$ $F1 = W'Y(Z' + X) + Z'X$ $F2 = WY'(Z + X') + ZX'$
19 compuertas
7 ANDs, 4 ORs, 2 XORs, 6 NOTs
Diseñador Humano 2
$F0 = (Z \oplus X)'(W \oplus Y)'$ $F1 = Z'X + (Z \oplus X)'(W'Y)$ $F2 = (F0 + F1)'$
13 compuertas
4 ANDs, 2 ORs, 2 XORs, 5 NOTs
MGA
$F0 = ((W \oplus Y) + (Z \oplus X))'$ $F1 = F2 \oplus ((W \oplus Y) + (Z \oplus X)$ $F2 = ((W \oplus Y) + (Z \oplus X))(((Z \oplus X) + (Z \oplus W)) \oplus X)$
9 compuertas
2 ANDs, 3 ORs, 3 XORs 2 NOTs
PG Prefija
$F0 = ((Z \oplus X) + Y') \oplus ((Z \oplus X) + W)$ $F1 = ((W + ((Z \oplus X) + Y')) \oplus (X' + Z)')'$ $F2 = ((Y'W \oplus Z) + (Z \oplus X)) \oplus X$
15 compuertas
1 AND, 5 ORs, 5 XORs 4 NOTs

Tabla 6.9: Comparación de las mejores soluciones obtenidas por MGA, la PG Prefija y dos diseñadores humanos para el tercer ejemplo: un comparador con 4 entradas 3 salidas.

6.3 Análisis de resultados

A continuación se compararán algunos resultados obtenidos por la PG Prefija y otras técnicas de diseño de circuitos. Se seleccionaron algunos resultados de las corridas más representativas para cada ejemplo. En este análisis no se presenta una corrida donde se describa el proceso de evolución de los circuitos, como se realizó en el capítulo de circuitos de una salida, debido principalmente a que resulta muy complicado describir lo que ocurre en cada una de las salidas, ya que éstas evolucionan en forma independiente.

6.3.1 Ejemplo 1

Corrida 1

La corrida 1 encuentra un circuito con 8 compuertas lógicas que puede ser comparado con el producido por el Diseñador Humano (DH), el cual tiene 12 compuertas. Las salidas $F0$ y $F2$ son iguales en ambos resultados. La salida $F1$ del DH se compone de 6 compuertas y la producida por la PG Prefija es de 3. Analizando ambas salidas, se observan componentes equivalentes obteniendo una reducción algebraica de la salida $F1$ del DH como se muestra a continuación:

$$(Z \oplus X)Y' + ((Z \oplus X) \oplus W)Y = (X \oplus Z) \oplus WY$$

Generando una nueva equivalencia lógica que puede ser de utilidad en la reducción algebraica, tenemos:

$$AB' + (A \oplus C)B = A \oplus CB$$

Donde

$$\begin{array}{rcl} A & = & Z \oplus X \\ B & = & Y \\ C & = & W \end{array}$$

Con esta equivalencia es posible reducir el circuito del DH a 8 compuertas lógicas

$$\begin{array}{llcl} & \text{PG Prefija} & & \text{Diseñador Humano} \\ F0 & W \oplus Y & = & W \oplus Y \\ F1 & (X \oplus Z) \oplus WY & = & (Z \oplus X)Y' + ((Z \oplus X) \oplus W)Y \\ F2 & ZX + WY(Z + X) & = & ZX + WY(Z + X) \end{array}$$

Corrida 6

La solución obtenida por la PG Prefija en la corrida 6 es idéntica a la obtenida por el BGA. El número de componentes, la reutilización de compuertas e iteraciones necesaria para llegar a la solución fue el mismo $(400,000)$.

$$\begin{array}{llcl} & \text{PG Prefija} & & \text{BGA} \\ F0 & W \oplus Y & = & W \oplus Y \\ F1 & WY \oplus (X \oplus Z) & = & (X \oplus Z) \oplus WY \\ F2 & WY(Z+X)+XZ & = & ZX+WY(Z+X) \end{array}$$

Corridas 1 y 6

Al igual que en los circuitos de una salida, para los circuitos de salidas múltiples se crean varias soluciones para un mismo resultado. La complejidad de las soluciones también puede variar, pero no es tan notorio como en los de una salida. En el problema de múltiples salidas, por lo general se tienen resultados que convergen a una solución única[1]. Tal es el caso de las corridas 1 y 2, en las que la salida $F0$ no varía nunca en su resultado. Esto es debido a lo sencillo que resulta la combinación de salida.

6.3.2 Ejemplo 2

Corrida 10

La solución del Diseñador Humano (de 8 compuertas) es equivalente a la encontrada por la PG Prefija en la corrida 10.

$$\begin{array}{llcl} & \text{PG Prefija} & & \text{Diseñador Humano} \\ F0 & WY & = & WY \\ F1 & XW \oplus ZY & = & WX \oplus ZY \\ F2 & (WY)'XZ & = & ZX(WY)' \\ F3 & WYXZ & = & WYXZ \end{array}$$

Corrida 16

La corrida 16 de la PG Prefija logra encontrar la misma solución que el MGA con 7 compuertas. La única diferencia es que requirió un menor esfuerzo en el número de iteraciones que el MGA: 325,000 evaluaciones de la función de aptitud contra 120,000 de la PG Prefija.

[1] Al hablar de general nos referimos a los 3 ejemplos que tomamos de la literatura y podría ser aplicado a circuitos donde la complejidad es mínima.

	PG Prefija		MGA
$F0$	WY	$=$	WY
$F1$	$XW \oplus ZY$	$=$	$WX \oplus ZY$
$F2$	$(WYXZ) \oplus XZ$	$=$	$ZX \oplus (WYZX)$
$F3$	$WYXZ$	$=$	$WYXZ$

Corrida 19

La PG Prefija encontró una solución de 9 compuertas lógicas en la corrida 19. Las salidas $F0$, $F2$, y $F3$ son equivalentes a las encontradas en la corrida 16 a excepción de la salida F1 que presenta una configuración de compuertas distinta. Vale la pena analizar dicha configuración, ya que si se realiza una simplificación algebraica es posible reducirla a 7 compuertas lógicas como se muestra a continuación:

$$F1 = XW \oplus (Z \oplus Y) \oplus (Y + Z)$$

Si aplicamos la ley asociativa a la expresión

$$XW((Z \oplus Y) \oplus (Y + Z))$$

Sabiendo que la expresión final es $XW \oplus ZY$ con base en el resultado de la salida $F1$ de la corrida 16, se obtiene una nueva equivalencia:

$$(Z \oplus Y) \oplus (Y + Z) = ZY$$

La nueva equivalencia fue verificada y es cierta, lo que nos conduce a generar la misma expresión que la obtenida en la salida $F1$ de la corrida 16, logrando reducir el circuito a 7 compuertas:

$$XW \oplus (Z \oplus Y) \oplus (Y + Z) = XW \oplus ZY$$

Las soluciones producidas por la PG Prefija son similares en componentes y pueden ser reducidas si se realiza un proceso de análisis algebraico:

	Corrida 16		Corrida 19
$F0$	WY	$=$	WY
$F1$	$XW \oplus ZY$	$=$	$XW \oplus (Z \oplus Y) \oplus (Y + Z)$
$F2$	$(WYXZ) \oplus XZ$	$=$	$ZX \oplus (WYZX)$
$F3$	$WYXZ$	$=$	$WYXZ$

Corridas 16 y 19

Las soluciones generadas por la PG Prefija pueden variar en complejidad. Sin embargo, si se realiza un análisis más detallado de los resultados, es posible encontrar expresiones de distinta configuración que, tras una análisis algebraico nos conducirá a encontrar equivalencias y, por consiguiente, a reducciones lógicas.

6.3.3 Ejemplo 3

Corrida 1

La solución de 19 compuertas del Diseñador Humano 1 es similar a una producida por la PG Prefija en la corrida 1 con igual número de compuertas. Las salidas $F1$ en ambos circuitos no son similares y no se encontró alguna nueva equivalencia lógica:

$$\begin{array}{lllll} & \text{PG Prefija} & & \text{Diseñador Humano 1} \\ F0 & (Y \oplus W + X \oplus Z)' & = & (Y \oplus W)'(X \oplus Z)' \\ F1 & ((WY' \oplus Z) + X) + ((W'Y)'Z') & = & W'Y(Z' + X) + Z'X \\ F2 & WY'(Z + X') + ZX' & = & WY'(Z + X') + ZX' \end{array}$$

Corrida 18

La solución propuesta por el Diseñador Humano 2 (DH2) supone la suma de $(F0 + F1)'$, lo que lo conduce a una solución de 13 compuertas. En la PG Prefija se encontró una solución en la corrida 18 con 18 compuertas. Si aplicamos lo propuesto por el DH2, entonces no tomamos en cuenta la salida $F2$ del circuito de la PG Prefija y lo reemplazamos por la suma y la negación de las salidas $F0$ y $F1$. Esto nos conduce a una solución de 12 compuertas.

Esta salida no fue propuesta como factible mínima debido a que se tendría que saber lo propuesto por el DH2 para poder hacer el reemplazo antes indicado.

$$\begin{array}{lllll} & \text{PG Prefija} & & \text{Diseñador Humano 2} \\ F0 & (W' \oplus Y) + (X \oplus Z)' & = & (Y \oplus W)'(X \oplus Z)' \\ F1 & (Z \oplus X)' \oplus ((W + Y) \oplus X \oplus W)X & = & Z'X + (Z \oplus X)'(W'Y) \\ F2 & (F0 + F1)' & = & (F0 + F1)' \end{array}$$

Capítulo 7

Conclusiones

Las regiones del espacio de diseño cubiertas por técnicas tradicionales resultan bastante restringidas en el caso de diseño de circuitos lógicos a comparación de las técnicas evolutivas, como lo son los algoritmos genéticos y la programación genética prefija, entre otras. El paralelismo implícito de las técnicas evolutivas permite explorar una región del espacio de diseño más amplia y de manera más eficiente.

Las expresiones prefijas cuentan con una característica implícita muy especial para el diseño de circuitos lógicos combinatorios: son fáciles de implementar y evaluar, puesto que no se requiere de mapeos adicionales. Además, la recombinación que experimenta la programación genética, permite explorar diversas regiones del espacio de diseño. Ambas características nos conducen a reducciones considerables del esfuerzo de cómputo y del tiempo de respuesta en comparación con otras técnicas alternativas.

La eficiencia en la solución de circuitos resulta aceptable, pudiéndose encontrar soluciones lo bastante robustas como para ser consideradas como buenas soluciones, que en ocasiones llegan a superar a otras técnicas.

Los parámetros utilizados en los experimentos muestran que es viable contar con una población pequeña, pero suficientemente amplia como para que nos permita disponer de una adecuada diversidad en la población. En el caso de circuitos de una salida se encontró que poblaciones de entre 90 y 500 individuos resultan eficientes, y para circuitos de múltiples poblaciones de 300 a 800 individuos.

El número de generaciones nos permite acceder a una recombinación de material genético "útil" y realizar una exploración más amplia del espacio de diseño. De acuerdo a nuestros experimentos, encontramos que un número máximo de generaciones de entre 100 y 1400 resultó aceptable para todos los circuitos de una salida analizados y de entre 400 y 2000 para los circuitos de multiples salidas.

Desafortunadamente los parámetros de tamaño de la población y número de generaciones van ligados a los problemas de dimensionalidad y complejidad del circuito que se desee encontrar. De esta forma tenemos circuitos del mismo número de variables de entrada, pero con una configuración de salida distinta que requieren diferentes parámetros de ejecución.

En los circuitos de una salida se encontró que resultan ser muy sencillos de resolver por la técnica propuesta debido a dos razones:

1. Cada individuo (expresión prefija) se especializa sólo en su resultado y no se espera una reutilización de código. El objetivo es tratar de encontrar al individuo más apto y cuyo número de compuertas sea menor.

2. El proceso de elitismo ejerce una presión de selección a tal grado que obliga a conservar a los individuos de menor tamaño.

La creatividad, principalmente expresada en los circuitos de una salida, resulta muy interesante ya que se generan soluciones robustas de distinto grado de complejidad, número y tipo de componentes; es decir, se obtienen circuitos robustos que cumplen satisfactoriamente con una salida en particular con el mismo número de compuertas, pero de distinto tipo o distinta configuración. Esta pequeña evidencia, nos hace pensar en lo complicado y vasto que resulta ser el espacio de diseño, ya que no se puede asegurar que se tenga una solución única lo bastante robusta.

Además, se observó el potencial de la PG Prefija para "descubrir" nuevas equivalencias algebraicas. Se realizó un análisis de los resultados obtenidos y se encontraron algunas nuevas equivalencias que podrían ser muy útiles en la simplificación por otro sistema adicional, como uno basado en casos. Respecto a este último punto se ha realizado ya algún trabajo relacionado [55], aunque usando como base un sistema que utiliza algoritmos genéticos para diseñar circuitos.

Ciertamente el problema de múltiples salidas es complejo y demuestra las limitantes de la implementación adoptada en este trabajo (la PG Prefija). En los resultados, se tienen soluciones robustas para el sumador de 2 bits y para el multiplicador de 2 bits, con parámetros razonables. Sin embargo, no ocurre lo mismo para el tercer circuito, ya que resulta muy complicado y es difícil encontrar ocurrencias de código que nos lleven a soluciones compactas.

Esta limitante de la técnica para llegar a soluciones óptimas es debido a que cada árbol dentro del bosque de árboles, evoluciona en forma independiente y sólo se espera que las ocurrencias de código emerjan. Sin embargo, nunca se realiza una conexión estrecha entre los árboles del bosque. De tal forma, es virtualmente imposible encontrar soluciones óptimas. Podemos intercambiar fragmentos de código entre los árboles, pero la posibilidad de colocarlo en el lugar adecuado es baja.

Claro que nuestro enfoque suele ser capaz de encontrar soluciones factibles. Sin embargo, debido a la presión de selección tan fuerte del elitismo, que preserva circuitos de menor tamaño y a la función de aptitud tan sencilla, que sólo busca la concordancia en salidas sin tomar en cuenta aspectos tan importantes como el tamaño del circuito, la similitud de código reutilizable y la maximización de conexiones, resulta casi imposible que se preserven estas buenas soluciones. De tal forma, conforme evolucionan las poblaciones, estas soluciones factibles tienden a desaparecer.

Como es claro, uno de los objetivos sería tratar de maximizar la cantidad de conexiones entre los árboles, tratando de encontrar módulos que nos permitan reutilizar código. Pensando en ésto, se analizó la propuesta de poder implantar ADFs (Funciones Automáticamente Definidas) [45, 46] o MA (adquisición de módulos) [43].

Finalmente se optó por no implementar ninguna de las dos alternativas. Las ADFs implicarían la creación de un número p de módulos encapsulados al inicio del programa. Sin embargo, este número no puede ser tan grande, pues consumiría muchos recursos de memoria. Por otro lado, la MA tiene la misma limitante que las ADFs. Además implicarían el agregar estos nuevos módulos al conjunto de funciones, ya que contarían con parámetros. El problema con ambas alternativas es que no es posible identificar la cantidad de módulos que se necesitarán para un circuito y resultaría más lógico pensar que esas conexiones se realicen en forma física, conectando nodos.

El espacio de diseño es amplio, pero cuando se habla de circuitos de múltiples salidas, las regiones del espacio se amplían aun más. Sin embargo, se podría diseñar eficientemente circuitos de gran complejidad tomando en cuenta un aspecto tan importante como la reutilización de código.

Finalmente, en la mayoría de los problemas se encontraron soluciones robustas que permiten observar la clara eficiencia de la técnica, así como sus limitaciones.

7.1 Trabajos futuros

Los distintos caminos a seguir en un futuro son:

- La construcción de una función de aptitud más robusta y versátil sería una de las primeras adaptaciones a realizar. El objetivo que se debe perseguir es llegar primero a la zona de diseños funcionales (osea, soluciones factibles) y posteriormente comenzar a preocuparse por el tamaño de las expresiones y número de compuertas.

- La reutilización y conexión de código "útil" es primordial para poder lograr una extensión exitosa a la región de múltiples salidas. Esto se podría lograr explorando tres posibilidades:

  1. Implementar una representación similar a la programación genética cartesiana, propuesta por Miller *et al.* [53], pero sin perder los beneficios que ofrece la representación prefija. Esto sería algo similar a utilizar pilas dinámicas, en lugar de las pilas estáticas que actualmente se emplean.
  2. Utilizar una nueva representación para el nodo raíz distinta a la de los demás nodos, que pueda adaptarse al número de salidas que se pretende diseñar y así satisfaga todas las posibles evaluaciones que se realicen con ese árbol; es decir ya no contar con un bosque de árboles, sino un solo árbol que satisface múltiples salidas.
  3. Crear algún mecanismo que permita la creación de módulos reutilizables, pero en forma más versátil y adecuada que permita evitar sobrecargas de memoria y uso excesivo de recursos de procesamiento. Sería una versión similar a las ADF's de Koza[46] y los MA de Angeline [1].

- Implementar una nueva función de aptitud que permita emplear técnicas multiobjetivo, con el fin de poder llevar a cabo la evolución del circuito tomando en cuenta la funcionalidad, el número de compuertas, las reconexiones y la utilización de módulos. Esto nos podría llevar a un plano más ambicioso que sería poder crear circuitos a nivel de funciones, utilizando nuestra técnica propuesta.

- Desarrollar un híbrido de esta técnica con un sistema basado en casos que permita la exploración y desarrollo de nuevas reglas algebraicas, utilizando los beneficios implícitos que arroja como resultado la PG Prefija, al usar expresiones-S que son propias de Prolog y Lisp.

Bibliografía

[1] Peter. J. Angeline. Genetic Programming and Emergent Inteligence. In Jr. Kenneth E. Kinnear, editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. The MIT Press, Cambridge, Massachusetts, 1994.

[2] Peter. J. Angeline. An Investigation into the Sensitivity of Genetic Programming to the Frequency of Leaf Selection During Subtree Crossover. In J.R. Koza, D.A. Goldberg, D.B. Fogel, and R.L. Riolo, editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 21–29. MIT Press, 1996.

[3] James Edward Baker. Reducing Bias and Inefficiency in the Selection Algorithm. In John J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 14–22. Lawrence Erlbaum Associates, Hillsdale, New Jersey, July 1987.

[4] E. Baldwin. *Genética Elemental*. Limusa, 1983.

[5] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming An Introduction*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998. On the Automatic Evolution of Computer Programs and Its Applications.

[6] Peter Bienert. Aufbau einer optimierungsautomatik für drei parameter. Master's thesis, Universidad Técnica de Berlin, 1967.

[7] George Boole. *An Investigation of the Laws of Thought*. Dover Pub., New York, 1954. Publicado en 1854.

[8] Hans J. Bremermann. The evolution of inteligence. Technical Report 477(17), Department of Mathematics, University of Washington, Seattle, July 1958.

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

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

[11] Carlos A. Coello, Alan D. Christiansen, and Hernández Aguirre. Using Genetic Algorithms to Design Combinational Logic Circuits. In Cihan H. Dagli, Metin Akay, C. L. Philip Chen, Benito R. Fernandez, and Joydeep Ghosh, editors, *ANNIE'96. Intelligent Engineering through Artificial Neural Networks*, volume 6 of *Smart Engineering Systems: Neural Networks, Fuzzy Logic and Evolutionary Programming*, pages 391–396, November 1996.

[12] Carlos A. Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Diseño Optimo de Circuitos Lógicos usando Algoritmos Genéticos. In *Primer Encuentro de Computación*, Taller de Aprendizaje, pages 1–10, Querétaro, Querétaro, Septiembre 1997.

[13] Carlos A. Coello, Christiansen Alan D., and Arturo Hernández Aguirre. Automated Design of Combinational Logic Circuits Using Genetic Algorithms. In D. G. Smith, N. C. Steele, and R. F. Albrecht, editors, *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms, ICANNGA'97*, pages 335–338, Norwich, England, April 1997. University of East Anglia.

[14] Carlos A. Coello Coello. *An Empirical Study of Evolutionary Techniques for Multiobjetive Optimization in Engineering Design.* PhD thesis, Department of Computer Science, Tulane University, New Orleans, Louisiana, April 1996.

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

[16] Carlos A. Coello Coello, Alan D. Christiansen, and Hernández Aguirre. Towards Automated Evolutionary Design of Combinational Circuits. *Computers and Electrical Engineering*, 27(1):1–28, January 2001.

[17] Carlos A. Coello Coello, Alan D. Christiansen, and Arturo Hernández Ãguirre. Use of Evolutionary Techniques to Automate the Design of Combinational Circuits. *International Journal of Smart Engineering System Design*, 2(4):299–314, June 2000.

[18] Carlos A. Coello Coello and Arturo Hernández Aguirre. Design of Combinational Logic Circuits through an Evolutionary Multiobjective Optimization Approach. Technical Report Lania-RI-2000-05, Laboratorio Nacional de Informática Avanzada, 2000.

[19] Carlos A. Coello Coello, Arturo Hernández Aguirre, and Bill P. Buckles. Evolutionary Multiobjective Design of Combinational Logic Circuits. In Jason Lohn, Adrian Stoica, Didier Keymeulen, and Silvano Colombano, editors, *Proceedings of the Second NASA/DoD Workshop on Evolvable Hardware*, pages 161–170, Los Alamitos, California, July 2000. IEEE Computer Society.

[20] 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 Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, pages 21–30, Edinburgh, Scotland, April 2000.

[21] 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 Lania-RI-2000-07, Laboratorio Nacional de Informática Avanzada, 2000.

[22] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J.J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Pittsburgh, PA., 1985. Carnegie-Mellon University.

[23] 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 1929).

[24] Lawrence Davis, editor. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[25] M. Dorigo and G. Di Caro. *The Ant Colony Optimization Meta-Heuristic.* McGraw-Hill, 1999. New Ideas in Optimization.

[26] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy. Technical Report 91-016, Dipartimento di Electtronica, Politecnico di Milano, Italy, 1991.

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

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

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

[30] Lawrence J. Fogel. *Artificial Intelligence through Simulated Evolution. Fourty years of Evolutionary Programming.* John Wiley & Sons, New York, 1999.

[31] George J. Friedman. Selective feedback computers for engineering synthesis and nervous system analogy. Master's thesis, University of California at Los Angeles, February 1956.

[32] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Alabama, USA, 1989.

[33] 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*, pages 69–93, San Mateo, California, 1989. Morgan Kaufmann.

[34] F. E. Hohn. *Applied Boolean Algebra.* The Macmillan Co., New York, second edition, 1966.

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

[36] E. V. Hunting. Sets of independent postulates for the algebra of logic. *Transactions American Mathematical Soc.*, 5:288–309, 1904.

[37] 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 Applications*, pages 260–275. Springer-Verlang, Berlin, Alemania, 1997.

[38] John Jenkins, editor. *Genetics.* Houghton Mifflin Company, Boston, Massachusetts, 1984.

[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] M. Karnaugh. A map method for synthesis of combinational logic circuits. *Transactions of the AIEE, Communications and Electronics*, 72(I):593–599, November 1953.

[42] Mike J. Keith and Martin C. Martin. Genetic Programming in C++: Implementation issues. In Jr. Kenneth E. Kinnear, editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. The MIT Press, Cambridge, Massachusetts, 1994.

[43] Jr. Kenneth E. Kinnear. Alternatives in automatic function definition: A comparison of performance. In Jr. Kenneth E. Kinnear, editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. The MIT Press, Cambridge, Massachusetts, 1994.

[44] John R. Koza. Herarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of 11th International Joint Conference on Artificial Intelligence*, pages 768–774, San Mateo, California, 1989. Morgan Kaufmann.

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

[46] John R. Koza. Scalable learning in genetic programming using automatic function definition. In Jr. Kenneth E. Kinnear, editor, *Advances in Genetic Programming*, chapter 5, pages 99–117. The MIT Press, Cambridge, Massachusetts, 1994.

[47] William B. Langdon. Evolving data structures with genetic programming. In Larry J. Eshelman, editor, *Preceedings of the Sixth International Conference on Genetic Algorithms*, pages 295–302. University of Pittsburgh, Morgan Kaufmann, July 1995.

[48] William B. Langdon. Using data structures within genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo,

editors, *Genetic Programming 1996: Preceedings of the First Annual Conference*, pages 141–148, Cambridge, MA., 1996. Stanford University, MIT Press.

[49] M. Morris Mano. *Arquitectura de Computadoras*. Prentice Hall, México, D.F., 1994.

[50] E. J. McCluskey. Minimization of boolean functions. *Bell Systems Technical Journal*, 35(6):1417–1444, November 1956.

[51] Gregor Johann Mendel. Experiments in plant hybridisation. *Journal of Royal Horticultural Society*, 26:1–32, 1901. Traducción al inglés de un artículo publicado originalmente en 1865.

[52] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, third edition, 1996.

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

[54] Julian F. Miller, P. Thomson, and T. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 105–131. Morgan Kaufmann, Chichester, England, 1998.

[55] Eduardo Islas Pérez. Development of a learning platform using case-based reasoning and genetic algorithms. case study: Optimization of combinational logic circuits. Master's thesis, Maestría en Inteligencia Artificial-UV/LANIA, Xalapa, Veracruz México, Novembre 2000.

[56] W. V. Quine. A way to simplify truth functions. *American Mathematical Monthly*, 62(9):627–631, November 1955.

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

[58] Charles H. Roth. *Fundamentals of Logic Design*. West Publishing Company, 4th edition, 1992.

[59] Tsutomu Sasao, editor. *Logic Synthesis and Optimization*. Kluwer Academic Press, 1993.

[60] Hans-Paul Schwefel. Kybernetische evolution als strategie der experimentall forschung in der strömungstechnik. Master's thesis, Universidad Tecnica de Berlin, 1965.

[61] Hans-Paul Schwefel and G. Rudolph. Contemporary evolution strategies. In *Advances in Artificial Life*, pages 893–907. Spinger-Verlag, Berlin, Alemania, 1995.

[62] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the AIEE*, 57:713–723, 1938.

[63] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning.* The MIT Press, Cambridge, Massachusetts, 1998.

[64] Gilbert Syswerda. Uniform Crossover in Genetic Algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9, San Mateo, California, 1989. Morgan Kaufmann.

[65] Herbert Taub. *Circuitos digitales y microprocesadores.* Mc Graw Hill, México, D.F., 1983.

[66] Ronald J. Tocci. *Sistemas Digitales principios y aplicaciones.* Prentice Hall, México, D.F., 1987.

[67] Katya Rodriguez Vazquez. *Multiobjetive Evolutionary Algorithms in Non-Linear System Identification.* PhD thesis, University of Sheffield, Sheffield, UK, 1999.

[68] August Weismann, editor. *The Germ Plasm: A Theory of Heredity.* UK, Scott, London, UK, 1893.

[69] A. Wetzel. *Evaluation of Efectiveness of genetic algorithms in combinational optimization.* University of Pittsburgh, Pittsburgh (unpublished), 1983.

[70] J. E. Whitesitt. *Boolean Algebra and its Applications.* Addison-Wesley Pub Co., Reading, Massachusetts, 1961.

[71] Darrel Whitley. The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, San Mateo, California, July 1989. Morgan Kaufmann Publishers.