
Centro de Investigación en Matemáticas, A.C.

# Síntesis de Funciones Lógicas Mediante un EDA Poliárbol.

T E S I S

que para obtener el grado de

Maestro en Ciencias con Especialidad en Computación y Matemáticas Industriales

presenta

Cyntia Araiza Delgado

Director de Tesis

Dr. Arturo Hernández Aguirre

*Guanajuato, Gto.* *Octubre de 2007*

# Índice general

# Índice de figuras

# Índice de cuadros

# Agradecimientos

Quisiera comenzar agradeciendo al Dr. Arturo Hernández Aguirre por su apoyo y por el tiempo dedicado para conducir a buen término este trabajo. Agradezco también al Dr. Enrique Raúl Villa Diharce y al Dr. Rogelio Hasimoto Beltrán por su paciencia y disposición para revisar este trabajo.

Mi más profundo agradecimiento a CIMAT por brindarme el espacio y el equipo necesario para realizar mis estudios de maestría, en especial a los profesores que contribuyeron a mi crecimiento académico y personal.

Un agradecimiento especial a mi familia: a mis padres, Irma y Antonio, que me han apoyado siempre; a mi hermana Yazmina, ya que ella me enseño que todo se puede conseguir con esfuerzo y dedicación.

A mis amigos Cristina, Oyuky, Ana, Eder y Gabriel por brindarme su apoyo a la distancia.

A Gustavo, por brindarme su amor, apoyo y comprensión, y por darme la fuerza necesaria para seguir adelante en la etapa final de mi estancia en el CIMAT.

Agradezco a mis compañeros y amigos: Leonel, Sergio Ivvan, Roberto, Oscar, Oyuki, Sergio, Dan-El, Juanita, William, Selma, Paquito, Esteban, Miky, Josué, Memo, David, Hugo y Luz por todo lo que aprendí de ellos, intelectual y humanamente, y también por los buenos momentos que compartimos dentro y fuera de CIMAT.

Finalmente, doy las gracias a CONACyT y a CIMAT que por medio de sus programas de becas me brindaron los medios económicos para realizar mis estudios de maestría.

# Capítulo 1

# Introducción

La producción de circuitos electrónicos es un proceso que demanda cada vez diseños mejores y más complejos, debido a las tecnologías actuales de fabricación. Sin embargo, tradicionalmente en el proceso de diseño se han adoptado técnicas que utilizan un conjunto de reglas y principios que además de haber existido durante décadas dependen de la habilidad del diseñador y no aseguran llegar al diseño óptimo. El diseño de circuitos lógicos combinatorios (no secuenciales) es un problema particular del diseño de circuitos electrónicos que tiene una complejidad tal que ha permanecido como un problema de investigación abierto a lo largo del tiempo.

El objetivo principal en el proceso de diseño de circuitos lógicos combinatorios es encontrar la expresión que porporcione el comportamiento deseado y que además minimice cierto criterio. En este caso el criterio a minimizar es el número de compuertas necesarios para satisfacer una tabla de verdad. Para minimizar este criterio, se han utilizado diversas técnicas, entre ellas están las aplicadas por la mano del hombre usando los Mapas de Karnaugh, los cuales sirven para diseñar el circuito que cumple con la función lógica dada.

Algunas de las técnicas que han sido aplicadas al diseño de circuitos lógicos son las conocidas como *Algoritmos Evolutivos*, que tienen este nombre debido a que logran encontrar soluciones a problemas de alta complejidad simulando el proceso evolutivo. Tambien existen los llamados *Algoritmos de Estimacion de Distribucion* $EDA$, cuya diferencia con los algoritmos evolutivos es que sustituyen los operadores de cruza

y mutacion por un algoritmo que estima la distribución de los datos.

A la utilización de métodos evolutivos para el diseño de hardware se le conoce como *Hardware Evolutivo*, el cual se divide en dos grupos principales, la evaluación intrínseca (o en línea) que es cuando se trabaja directamente con hardware reconfigurable y la extrínseca (o fuera de línea) y es cuando únicamente se trabaja con simulaciones.

El objetivo de este trabajo de tesis es la implementación de los *Algoritmos de Estimacion de Distribución (EDAS)*, especificamente del EDA discreto conocido como *Poliárbol*, al *Hardware Evolutivo* para llevar a cabo la síntesis de funciones lógicas y poder diseñar un circuito lógico funcional y además óptimo.

La estructura de la tesis está organizada en seis capítulos, de los cuales a continuación se da un breve resumen de su contenido.

En el capítulo 1 se da una breve explicación del objetivo de esta tesis, terminando con una breve descripción de su contenido.

En el capítulo 2 se explica lo que es el *Hardware Evolutivo*, además de describir a detalle las compuertas utilizadas para la síntesis de las funciones lógicas.

La descripción de los *EDAs* se hace en el capítulo 3, y se incluyen también varios EDAs existentes, que se implementaron para la realización de pruebas, así como una comparación con el *Algoritmo Genetico*, con la finalidad de mostrar que ventajas ofrece la utilización de un algoritmo de estimación de distribución sobre un algoritmo genético.

En el capítulo 4 se hace una descripción a detalle del algoritmo implementado en este trabajo de tesis, el conocido *Poliárbol*, mostrando un pequeño experimento que se hizo inicialmente para probar la dependencia existente entre variables aleatorias binarias, y finalmente su aplicación al *Hardware Evolutivo*.

En el capítulo 5 se muestran los experimentos y resultados obtenidos al aplicar el EDA *Poliárbol* al *Hardware Evolutivo*. Se presentan las tablas de verdad correspondientes a cada problema planteado, así como

los paramétros utilizados para cada corrida del algoritmo, mostrando como resultado un circuito lógico combinatorio, y en el caso de algunos de los parametros se muestra una comparación con otros métodos.

En el capítulo 6 se dan las conclusiones de los experimentos realizados en este trabajo de tesis.

# Capítulo 2

# Hardware Evolutivo

## 2.1. Antecedentes

Un sistema digital es una combinación de dispositivos diseñada para manipular variables que solamente tomen valores discretos. Un *circuito lógico combinatorio* es un sistema digital que opera en modo binario, es decir que los valores que pueden existir sólo son el 1 y el 0, además de que en cualquier instante de tiempo la salida depende únicamente de los niveles lógicos presentes a la entrada.

Gracias a la computadora se han logrado los avances significativos de nuestros días. Su diseño, mantenimiento y análisis de operación se hace mediante técnicas y simbologías conocidas como álgebra de Boole [13]. Lleva este nombre en honor del matemático inglés George Boole, quien publicó en 1854 "Investigación de las leyes del pensamiento, sobre las que se basan las teorías matemáticas de la lógica y la probabilidad", donde Boole realizó un análisis matemático de la lógica. Fue C.E. Shannon [6] quien en 1938 utilizó el álgebra de Boole para representar el comportamiento de los circuitos de interruptores o de conmutación. Dada la similitud entre los circuitos de interruptores con los circuitos lógicos combinatorios, las técnicas usadas por Shannon son las mismas.

En la literatura moderna, se llama *Hardware Evolutivo* al uso de técnicas basadas en la evolución para el diseño de circuitos lógicos combinatorios.

Zebulum [6] propone una taxonomía del hardware evolutivo de acuer-

do a cuatro propiedades, las cuales son:

- Proceso de evaluación de hardware
- Enfoque de computación evolutiva
- Área de aplicación
- Plataforma evolutiva

### 2.1.1. Proceso de evaluación de Hardware

Una posible forma de definir el Hardware Evolutivo es como la integración de dispositivos reconfigurables con aprendizaje génetico. La propiedad de *proceso de evaluación de hardware* trata con el tipo de ambientes donde se evalúan cromosomas durante el aprendizaje genético. Se puede clasificar en dos ramas:

- Hardware Evolutivo Intrínseco
- Hardware Evolutivo Extrínseco

En el enfoque *Hardware Evolutivo Intrínseco*, la evaluación se lleva a cabo en el dispositivo reconfigurable. Cromosomas o genotipos son introducidos en el dispositivo reconfigurable y evaluados usualmente a través de los mecanismos de hardware. Hay dos ventajas en la utilización de este enfoque: la primera es la velocidad del proceso de evolución y la segunda y la más importante, el hecho de que no se necesita la simulación para evaluar el circuito. Por lo tanto, en este caso, todas las soluciones se comportarán exactamente como lo harían en una aplicación en tiempo real. Su posible desventaja sería el costo de los dispositivos reconfigurables, necesarios para la experimentación.

El enfoque *Hardware Evolutivo Extrínseco*, consiste en el uso de software para simular y evaluar los circuitos. Al final del proceso evolutivo, la solución encontrada (el mejor individuo) es introducida en el dispositivo reconfigurable. El enfoque extrínseco es más simple y en muchos casos, la única forma práctica de evaluar la aptitud de un número grande de individuos. Adicionalmente, este enfoque facilita la experimentación con nuevas teorías de Hardware Evolutivo.

### 2.1.2. Enfoque de Computación Evolutiva

El Hardware Evolutivo utiliza la evolución artificial para diseñar circuitos, pero la forma en la que los investigadores implementan sus algoritmos evolutivos puede variar. Hay básicamente tres diferentes enfoques:

- Algoritmos Genéticos Tradicionales
- Programación Genética
- Programación Evolutiva

Los *Algoritmos Géneticos Tradicionales* $AGs$ como fueron propuestos originalmente por John Holland, tienen las siguientes características básicas:

- Codificación Binaria para formar Cromosomas
- Cruza de un punto y mutación como operadores genéticos
- Reemplazamiento generacional de la población total con elitismo (seleccionando solo los mejores individuos)

La principal ventaja de la representación binaria es que ésta provee el número máximo de esquemas (bloques contructores) por unidad de información. Sin embargo, la representación binria puede implicar longitudes muy largas de cromosomas en algunas aplicaciones.

Además de los tres métodos básicos, hay otros enfoques que pueden ser aplicados al Hardware Evolutivo, uno de ellos son los *Algoritmos de Estimación de Distribución* $EDAs$, los cuales se explican en el siguiente capítulo de esta tesis.

### 2.1.3. Área de Aplicación

El Hardware Evolutivo comprende una variedad de aplicaciones que pueden ser clasificadas de acuerdo a las siguientes categorías:

- Diseño de Circuitos (digitales y analógicos)
- Control y robótica
- Reconocimiento de patrones
- Tolerancia a fallas

- VLSI

El *diseño de circuitos electrónicos*, tanto digitales como analógicos es la principal aplicación de la electrónica evolutiva. La disponibilidad de un gran rango de simuladores de circuitos para experimentos extrínsecos, así como dispositivos programables, como lo son los FPGAs usados en sistemas de Hardware Evolutivo intrínsecos, ha permitido a los investigadores estudiar los sistemas de hardware evolutivo.

En el diseño de circuitos, las estructuras evolutivas pueden ser transistores, compuertas de transmisión o capas. El emplazamietno y el ruteo son básicamente problemas de optimización combinatoria que pueden beneficiarse ampliamente de los enfoques evolutivos.

### 2.1.4. Plataforma Evolutiva

Esto se refiere a la plataforma de hardware sobre la cual se lleva a cabo la evolución del circuito. Se pueden concebir tres clases principales de Plataformas Evolutivas:

- Circuitos Integrados Programables
- Hardware Dedicado
- Diseño de Circuitos Integrados

Los *circuitos integrados programables* son la plataforma estándar utilizada en Hardware Evolutivo. Estos pueden ser divididos en tres categorías: *memorias, microprocesadores y circuitos lógicos*. El Hardware Evolutivo se enfoca principalmente en Circuitos Lógicos Programables. A su vez, los circuitos lógicos pueden ser divididos en tres sub-categorías: PLD (Programmable Logic Device), CPLD (Complex Programmable Logic Device) y FPGA (Field Programmable Gate Array).

Los circuitos PLD consisten en un arreglo de compuertas AND, el cual genera términos para las entradas del sistema y un arreglo de compuertas OR que genera la salida del sistema. De acuerdo a su grado de programabilidad, los circuitos PLD pueden ser clasificados en PROM (Programmable Read-Only Memory), PAL (Programmable Array Logic) y PLA (Programmable Logic Array).

Un CPDL puede verse como una combinación de celdas programables que consiste en normalmente de multiplexores o memorias y una red de interconexión que selecciona las entradas de las celdas programables.

Los FPGAs son los dispositivos reconfigurables más usados en el Hardware Evolutivo. Consisten en un arreglo de celdas lógicas y celdas I/O. Cada celda lógica consiste en una función universal (multiplexor, demultiplexor y memoria) que puede ser programada para realizar cierta funcion.

## 2.2. Compuertas Lógicas

La compuerta lógica es el bloque de construcción básico de los sistemas digitales, que constituye un bloque de hardware que puede activarse o desactivarse al satisfacerse los requerimientos lógicos de la entrada. A continuación se describirán las compuertas utilizadas para el diseño de los circuitos lógicos [8].

### 2.2.1. Compuertas Básicas

Existen tres compuertas básicas con las que se puede construir cualquier sistema digital, y estas son las compuerta OR, la compuerta AND y la compuerta NOT.

#### Compuerta OR

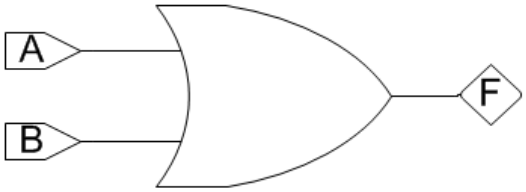
Circuito lógico de dos o más entradas, cuya salida es igual a la suma lógica de las entradas. La figura 2.1a muestra su tabla de verdad, en donde podemos ver que la salida será 1 siempre y cuando por lo menos una de las entradas tenga un valor de 1, y será 0 cuando todas las entradas sean 0. Su símbolo gráfico se muestra en la figura 2.1b.

#### Compuerta AND

La salida de la compuerta AND es igual al producto de sus entradas, las cuales pueden ser dos o más. Su representación gráfica se muestra en la figura 2.2b, y su tabla de verdad en la figura 2.2a, en donde podemos que ver que su salida es 1 sólo cuando todas sus entradas son 1.

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(a)

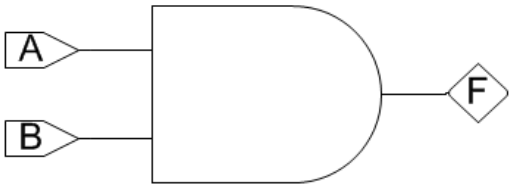



(b)

Figura 2.1: (a)Tabla de verdad de la compuerta OR. (b) Símbolo del circuito

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a)




(b)

Figura 2.2: (a)Tabla de verdad de la compuerta AND. (b) Símbolo del circuito

#### Compuerta NOT

También conocido como inversor, es un circuito lógico que tiene una sola entrada y una sola salida. La salida de este circuito siempre será el valor contrario al que entro. La tabla correspondiente a este circuito así como su símbolo gráfico se muestran en la figura 2.3.

### 2.2.2. Compuertas Universales

La compuerta NAND y la NOR son muy utilizadas para implementar diseños. Esto se debe a que cualquiera de estas dos compuertas puede ser usada para representar cualquiera de las tres compuertas básicas.

#### Compuerta NAND

Esta compuerta es equivalente a la compuerta AND seguida de la compuerta NOT, como se muestra en la figura 2.4b. La tabla de verdad

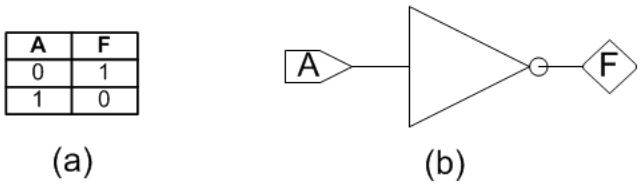



Figura 2.3: (a)Tabla de verdad de la compuerta NOT. (b) Símbolo del circuito

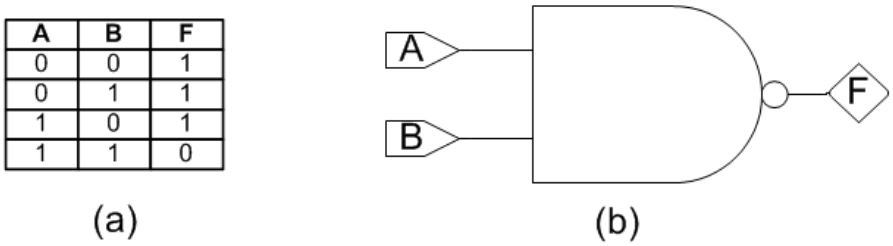



Figura 2.4: (a)Tabla de verdad de la compuerta NAND. (b) Símbolo del circuito

correspondiente a esta compuerta se muestra en la figura 2.4a, en donde podemos ver que su salida es 0 cuando todas sus entradas son 1.

#### Compuerta NOR

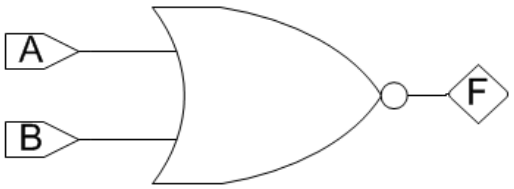
La compuerta NOR es una combinación de la compuerta OR y de la NOT. Su tabla de verdad y su representación gráfica se muestran en la figura 2.5. Observando la tabla de verdad podemos notar que para que el valor de salida de esta compuerta sea 1 todas sus entradas deberán ser 0.

### 2.2.3. Otras Compuertas

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

| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

(a)

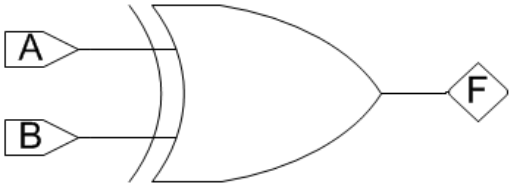



(b)

Figura 2.5: (a)Tabla de verdad de la compuerta NOR. (b) Símbolo del circuito

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

(a)




(b)

Figura 2.6: (a)Tabla de verdad de la compuerta XOR. (b) Símbolo del circuito

#### Compuerta OR Excluyente o XOR

La compuerta XOR tiene como función booleana la siguiente expresión:

$$X = A'B + AB' \tag{2.1}$$

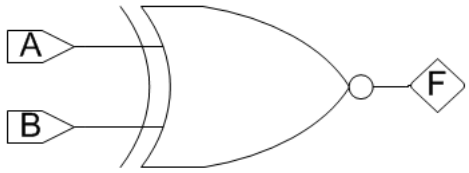
Su representación gráfica y su tabla de verdad se muestran en la figura 2.6. En la tabla de verdad podemos ver que su salida será 1 siempre y cuando una de sus entradas sea 1, en otras palabras, producirá una salida con valor 1 siempre que las entradas tengan valores opuestos.

#### Compuerta NOR Excluyente o XNOR

La negación de la compuerta XOR, como se muestra en la figura 2.7b, genera a la compuerta XNOR o también conocida como de equivalencia. El por qué de su nombre lo podemos observar en la tabla de verdad, la cual podemos ver en la figura 2.7a, en donde se muestra que la salida es 1 únicamente cuando ambas entradas son iguales.

| A | B | F |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a)




(b)

Figura 2.7: (a)Tabla de verdad de la compuerta XNOR. (b) Símbolo del circuito

## 2.3. Postulados, Leyes y Teoremas del Álgebra Booleana

El álgebra booleana trata principalmente con constantes, variables y operaciones lógicas y su principal diferencia con respecto al álgebra tradicional es que en este caso se utilizan variables y constantes binarias, es decir que solo pueden tomar el valor de 0 o de 1. El álgebra booleana cuenta solo con tres operaciones básicas [11]:

1. **Adición Lógica.** Conocida como operación OR. El símbolo que representa esta operación es el $(+)$. La salida es 1 cuando cualquiera de las entradas es 1.
2. **Multiplicación Lógica.** También conocida como operación AND. El símbolo es el $(\cdot)$. La salida será 1 cuando ambas entradas sean 1.
3. **Complemento.** Se le conoce también como operación NOT o inversora. Su símbolo es el $(')$. Y así como lo indica su nombre, la salida siempre será lo contrario al valor de la entrada.

Los postulados, leyes y teoremas booleanos sirven para poder simplificar las expresiones booleanas que representan un circuito lógico y en consecuencia requerir un menor número de compuertas para su implementación.

### 2.3.1. Postulados

Los postulados describen el comportamiento de las operaciones básicas del álgebra booleana.

1. a) $0 \cdot 0 = 0$
   b) $1 + 1 = 1$
2. a) $1 \cdot 1 = 1$
   b) $0 + 0 = 0$
3. a) $1 \cdot 0 = 0 \cdot 1 = 0$
   b) $0 + 1 = 1 + 0 = 1$
4. a) $0' = 1$
   b) $1' = 0$

### 2.3.2. Leyes

Las tres leyes básicas del álgebra booleana se presentan a continuación:

1. **Ley Asociativa.** Establece que las variables de una operación AND o de una operación OR pueden ser agrupadas en cualquier forma.
   a) $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
   b) $A + (B + C) = (A + B) + C$
2. **Ley Distributiva.** Establece que una expresión puede desarrollarse multiplicando término a término como en el álgebra ordinaria así como llevar a cabo la factorización de términos.
   a) $A \cdot B + A \cdot C = A \cdot (B + C)$
   b) $(A + B) \cdot (A + C) = A + B \cdot C$
3. **Ley conmutativa.** Esta ley establece que no importa el orden en que se evalúen las operaciones AND y OR de dos variables ya que el resultado siempre será el mismo.
   a) $A \cdot B = B \cdot A$
   b) $A + B = B + A$

### 2.3.3. Teoremas

Existen diversos teoremas que resultan de la aplicación de los postulado y de las leyes mencionadas anteriormente, sin embargo sólo se mostrarán los más útiles.

1. Teorema de la Identidad
   a) $A + 0 = A$
   b) $A \cdot 1 = A$
2. Teorema del Elemento Nulo
   a) $A + 1 = 1$
   b) $B \cdot 0 = 0$
3. Teorema del Complemento
   a) $A + A' = 1$
   b) $A \cdot A' = 0$
4. Teorema de la Idempotencia
   a) $A + A = A$
   b) $A \cdot A = A$
5. Teorema de la Absorción
   a) $A \cdot (A + B) = A$
   b) $A + A \cdot B = A$
6. Teorema de Involución
   a) $A = A''$
7. Teorema de DeMorgan
   a) $(A + B + C + ...)' = A' \cdot B' \cdot C' \cdot ...$
   b) $(A \cdot B \cdot C \cdot ...) = A' + B' + C' + ...$

## 2.4. Aplicaciones

En la actualidad un nuevo paradigma de computación ha venido desarrollándose: *los sistemas dinámicamente reconfigurables*. Estos sistemas utilizan hardware cuya funcionalidad puede modificarse en tiempo de ejecución. Este es el conocido Hardware Evolutivo, el cual brinda una gran flexibilidad sin comprometer el desempeño.

Estos sistemas reconfigurables combinan un componente reconfigurable con un procesador tradicional, y pueden ser configurados óptimamente para cada una de las tareas que conforman una aplicación,

cumpliendo con sus restricciones dentro de un mismo chip.

El Hardware Evolutivo ha adquirido relevancia en el campo de los procesadores de propósito general debido a su capacidad de reconfiguración para ejecutar diferentes tareas sin interrumpir su operación. Muchas arquitecturas reconfigurables de grano grueso están empleándose como coprocesadores reconfigurables, asumiendo la carga del procesador principal en las aplicaciones multimedia gracias a su alto nivel de paralelismo.

Igualmente, las características anteriores hacen de estas arquitecturas una alternativa inmejorable respecto a los procesadores digitales de señal (DSPs) y los circuitos integrados de aplicación específica (ASICs) para aplicaciones con alto grado de paralelismo de datos y requisitos computacionales de alto desempeño, tales como el procesamiento de señales, imágenes y video, multimedia y encriptación de datos.

La evolución de este tipo de sistemas ha impulsado simultáneamente el desarrollo de nuevas metodologías de diseño y herramientas CAD para la compilación y síntesis.

Además tiene una fuerte aplicación en lo que es la robótica, como lo podemos ver en *Henriette*, un robot con forma de gallina creado por un equipo de investigadores noruegos. Este robot usó hardware evolutivo sobre un software hecho para que aprendiese a caminar y a hablar solo.

En estos casos, el hardware puede resolver muchos problemas que no puede resolver el software. Por ejemplo, un programador informático no puede pensar en todos los problemas con los que se puede encontrar un robot recorriendo la superficie de Marte. Mediante el hardware evolutivo el robot puede aprender por él mismo y resolver una situación problemática sin la intervención humana.

# Capítulo 3

# Algoritmos de Estimación de Distribución

## 3.1. Introducción a los EDAs

El comportamiento de los algoritmos evolutivos depende de varios párametros asociados a ellos (operadores de cruza y mutación, tamaño de la población, probabilidades de cruza y mutación, número de generaciones, etc.). Si un usuario no tiene experiencia usando este tipo de métodos para la resolución de un problema concreto de optimización, entonces el ajuste de estos parámetros se convierte en otro problema de optimización.

Por esta razón, junto con el hecho de que la predicción de los movimientos de la poblacion en el espacio de búsqueda es extremadamente difícil, se motivó al nacimiento de un tipo de algoritmos conocidos como Algoritmos de Estimación de Distribución (EDAs) [10]. Los EDAs fueron introducidos en el campo de la computación evolutiva por primera vez por Mühlenbein y Paa$\beta$.

Holland [7] ya había reconocido que tomar en cuenta la interacción entre las variables podía resultar benéfico para los AGs. Esta inexplotada fuente de conocimiento fue llamada *información de enlace* (linkage information). Siguiendo esta idea, en otros métodos desarrollados por diferentes autores (Goldberg, Kargupta, Bandyopadhyay, Lobo, Kemenade, Bosman y Thierens) los algoritmos genéticos simples se extendieron al proceso de bloques constructores.

En los EDAs no existe ni el operador de cruza ni el de mutación. En lugar de esto, la nueva población se muestrea de una distribución de probabilidad, la cual es estimada de una muestra que contiene individuos seleccionados previamente de la población de la generación anterior. Mientras que en las heurísticas de computación evolutiva, las interrelaciones (bloques constructores) entre las diferentes variables que representan a los individuos se tienen en mente implícitamente, en los EDAs las interrelaciones son explícitamente expresadas a través de la unión de la distribución de probabilidad asociada con los individuos seleccionados en cada iteración. De hecho, la estimación de la unión de la distribución de probabilidad asociada con la muestra que contiene estos individuos seleccionados constituye un obstáculo para esta nueva heurística.

Hay tres pasos que constituyen a un EDA:

1. Seleccionar una muestra de una población.

2. Calcular una distribución de probabilidad para dicha muestra.

3. Regenerar la nueva población usando la distribución calculada.

Estos tres pasos se repiten hasta que un cierto criterio de paro previamente establecido se cumple. A continuación se muestra el algoritmo general de un EDA.

**Algoritmo 1** Algoritmo General de un EDA

**Inicializar:** $P_0 \leftarrow$ Generar $N$ individuos aleatoriamente

1: **repetir**
2: **para** $i = 1, 2, ...$ **hacer**
3: $S_{i-1} \leftarrow$ Seleccionar $M \leq N$ individuos de $P_{i-1}$ usando un método de selección (MUESTRA)
4: $p_i(x) = p(x|S_{i-1}) \leftarrow$ Estimar la distribución de probabilidad de la muestra seleccionada
5: $P_i \leftarrow$ Regenerar $N$ individuos (de la nueva población) utilizando $p_i(x)$
6: **fin para**
7: **mientras** que cierto criterio de paro no se cumple

## 3.2. El EDA en la Optimización Combinatoria

### 3.2.1. Sin Dependencias.

En todos los métodos que pertenecen a esta categoría se asume que la distribución de probabilidad n-dimensional se factoriza como el producto de n distribuciones de probabilidad univariadas o independientes. Esto es $p_i(x) = \prod_{j=1}^{n} p_i(x_j)$. Obviamente asumir esto está muy lejos de lo que realmente sucede en un problema de optimización con cierta dificultad, donde las dependencias entre las variables usualmente si existen.

Entre los algoritmos univariados más comunes se encuentran el UMDA y el PBIL, los cuales se explican a continuación.

#### UMDA

Creado por Mühlenbein. El algoritmo de distribucuón marginal univariada usa el modelo consiste en estimar la distribución de probabilidad $p_i(x)$ de los individuos seleccionados en cada generación. Y esta distribución es factorizada como el producto de distribuciones marginales univaradas independientes. Esto es:

$$p_i(x) = p(x|S_{i-1}) = \prod_{j=1}^{n} p_i(x_j)$$

Cada distribución marginal univariada es estimada de frecuencias marginales:

$$p_i(x_j) = \frac{\sum_{k=1}^{M} \delta_k(X_j = x_j|S_{i-1})}{M}$$

donde

$$\delta_k(X_j = x_j|S_{i-1}) = \begin{cases} 1 & \text{si el j-ésimo caso de } S_{i-1} \text{ , } X_j = x_j \\ 0 & \text{otro caso} \end{cases}$$

El algoritmo 2 muestra la forma general del UMDA.

#### PBIL

El PBIL (Población Basada en Aprendizaje Incremental) fue introducido por Baluja, y después fue probado por Baluja y Caruana, con el objetivo de obtener el óptimo de una función definida en el espacio

**Algoritmo 2** Algoritmo General del UMDA

**Inicializar:** $P_0 \leftarrow$ Generar $N$ individuos aleatoriamente

1: **repetir**
2: **para** $i = 1, 2, ...$ **hacer**
3: $S_{i-1} \leftarrow$ Seleccionar $M \leq N$ individuos de $P_{i-1}$ usando un método de selección (MUESTRA)
4: $p_i(x) = p(x|S_{i-1}) \leftarrow p_i(x_j) = \frac{\sum_{k=1}^{M} \delta_k(X_j = x_j|S_{i-1})}{M}$
5: $P_i \leftarrow$ Regenerar $N$ individuos (de la nueva población) utilizando $p_i(x)$
6: **fin para**
7: **mientras** que cierto criterio de paro no se cumple

binario $\Omega = \{0, 1\}^n$. En cada generación, la población de individuos es representada por un vector de probabilidades:

$$p_i(x) = (p_i(x_1), ..., p_i(x_j), ..., p_i(x_n))$$

El algoritmo 3 muestra la forma general del PBIL.

**Algoritmo 3** Algoritmo General del PBIL

1: Obtener el vector inicial de probabilidad $p_0(x)$
2: **repetir**
3: Usando $p_i(x)$ generar $N$ individuos: $x_1^i, ..., x_k^i, x_N^i$
4: Evaluar y ordenar $x_1^i, ..., x_k^i, x_N^i$
5: Seleccionar $M(M \leq N)$ de los mejores individuos: $x_{1:N}^i, ..., x_{k:N}^i, x_{M:N}^i$
6: Actualizar el vector de probabilidad $p_{i+1}(x) = (p_{i+1}(x_1), .., p_{i+1}(x_n))$
7: **para** $i = 1, ..., n$ **hacer**
8: $p_{i+1}(x_j) = (1 - \alpha)p_i(x_j) + \alpha \frac{1}{M} \sum_{k=1}^{M} x_{j,k:N}^i$
9: **fin para**
10: **mientras** que cierto criterio de paro no se cumple

### 3.2.2. Dependencias Bivariadas.

Estimar la distribución de probabilidad se hace de manera más rápida si asumimos independencia entre las variables, lo cual está muy alejado de la realidad en algunos problemas, caso contrario a si tomamos en cuenta dependencias entre cada par de variables. En este caso, es suficiente considerar estadísticos de segundo orden. Mientras que en los algoritmos univariados, el aprendizaje de sus parámetros fue considerado, en este tipo de algoritmos, el aprendizaje paramétrico es extendido al aprendizaje estructurado también. Entre los métodos bivariados más conocidos se encuentran el MIMIC y el BMDA, de los cuales se hablará mas detalladamente más adeltante.

### 3.2.3. Dependencias Múltiples.

Varios de los algoritmos EDAs que se han propuesto en la literatura requieren una factorización de la distribución de la probabilidad con estadísticos de orden mayor a 2. Hasta donde sabemos, el primer trabajo en el cual la posibilidad de adaptación de los métodos de modelos de inducción desarrollados por la comunidad científica el que trabaja sobre modelos probabilísticos gráficos para EDAs, es el de Baluja y Davies.

Esta posibilidad es mencionada de nuevo en un trabajo posterior, pero desafortunadamente solo se mencionó y no hay evidencia de implementación.

Entre los métodos con dependencias múltiples más conocidos se encuentran el PADA y el BOA.

#### PADA

En este método la factorización es hecha usando una red bayesiana con estructura de poliárbol (no más de un arista conecta a cada par de variables). El algoritmo propuesto se llama PADA (Algoritmo de Aproximación de Distribución basado en Poliárbol) y puede ser considerado un híbrido entre un método de detección de dependencias e independencias condicionales y un procedimiento basado en aptitud y búsqueda.

Este método se explica a detalle en el capítulo 4 de esta tesis.

#### BOA

Este algoritmo fue propuesto por Pelikan y Goldberg. El BOA (Algoritmo de Optimización Bayesiano) es un algoritmo que usa el BDe (equivalencia Bayesiana de Dirichlet), que es una métrica de medida de bondad para cada estructura. Esta métrica Bayesiana tiene la propiedad de que el puntaje de dos estructuras que reflejan la misma dependencia o independencia condicional es el mismo.

La búsqueda utilizada es una *búsqueda glotona* y comienza en cada generación desde el principio. Con el fin de reducir la cardinalidad del espacio de búsqueda se asume la restricción de que cada nodo de la red Bayesiana tiene al menos $k$ padres.

## 3.3. Algoritmo MIMIC

El MIMIC es un algoritmo de optimización cuyas siglas significan Mutual Information Maximizing Input Clustering [5]. Este algoritmo busca en cada generación la mejor permutación entre las variables con el fin de encontrar la distribución de probabilidad $p_l^{\pi}(x)$, esto es lo más cercano a la distribución empirica de la muestra seleccionada cuando se usa la distancia de Kullback-Liebler.

Hay dos componentes principales en el MIMIC, el primero es un algoritmo de optimización aleatorio que muestrea aquellas regiones del espacio de las variables que muy probablemente contengan el mínimo de la función que se busca optimizar; segundo, un estimador de densidad efectivo que puede ser usado para capturar una gran variedad de la estructura del espacio de las variables, ésto aún es computable para simples estadísticos de segundo orden, y tiene un mejor desempeño que las aproximaciones relativas.

Sea $C(x)$ la función que se desea optimizar, también conocida como función de costo. Conociendo solo $C(x)$ parece lógico buscar sus mínimos en cada generación de puntos con una distribución uniforme sobre las entradas $p(x)$. Tal búsqueda nos impide que algún punto de información generada por muestras previas tenga efecto alguno en la generación de las muestras subsecuentes. No es de sorprender que una cantidad menor de trabajo sea necesaria si las muestras fueran generadas de una distribución, $p^{\theta}(x)$, la cual está uniformente distribuida sobre aquellas $x's$ donde $C(x) \leq \theta$ y tiene una probabilidad de $\phi$ en otro lado. Por ejemplo, si tuvieramos acceso a $p^{\theta_m}(x)$ para $\theta_m = min_x C(x)$, una simple muestra podría ser suficiente para encontrar un óptimo.

Este conocimiento sugiere un proceso de aproximación sucesiva: dada una colección de puntos para los cuales $C(x) \leq \theta_0$ se construye un estimador de densidad para $p^{\theta_0}(x)$. Se generan nuevas muestras con este estimador de densidad adicional, y se establece un nuevo umbral, $\theta_1 = \theta - \epsilon$ , y se actualiza el estimador de densidad. Este proceso se repite hasta que los valores de $C(x)$ dejen de mejorar.

El algoritmo MIMIC comienza generando una población aleatoria de candidatos elegidos uniformemente del espacio de entrada. Se extrae la

mediana de la aptitud de esta población, la cual está denotada por $\theta_0$. A continuación se muestra el algoritmo MIMIC.

1. Actualizar los parámetros del estimador de densidad de $p^{\theta_i}(x)$ para una muestra.
2. Generar más muestras de la distribución $p^{\theta_i}(x)$.
3. Fijar $\theta_{i+1}$ igual al enésimo percentil de los datos. Retener sólo los puntos menores a $\theta_{i+1}$.

La validez de esta aproximación depende de dos suposiciones críticas:

- $p^{\theta}(x)$ puede ser aproximada exitosamente con una cantidad finita de datos.
- $D(p^{\theta-\epsilon}(x)\|p^{\theta}(c))$ es lo suficientemente pequeña tal que las muestras de $p^{\theta}(x)$ son además probables de ser muestras de $p^{\theta-\epsilon}(x)$ (donde $D(p\|q)$ es la divergencia de Kullback-Liebler entre p y q).

Los límites sobre estas condiciones pueden ser usados para probar convergencia en un número finito de pasos de aproximación exitosa. El desempeño de esta aproximación depende de la naturaleza del aproximador de densidad usado.

### 3.3.1. Generando Eventos de Probabilidades Condicionales

La unión de la distribución de probabilidad sobre un conjunto de variables aleatorias $X = X_i$, es:

$$p(X) = p(X_1|X_2, ..., X_n)p(X_2|X_3, ..., X_n)...p(X_{n-1}|X_n)p(X_n)$$

En la figura 3.1 podemos ver la estructura de este algoritmo.

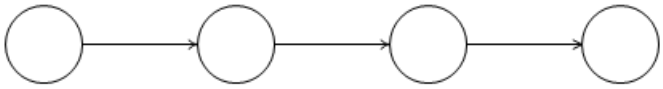


Figura 3.1: Estructura del MIMIC

Dado que tenemos probabilidades condicionales, $p(X_i|X_j)$, y probabilidades marginales, $p(X_i)$, se tiene la tarea de generar muestras que

se emparejen tan cerca como sea posible a la unión verdadera de distribución, $p(X)$.No es posible capturar todas las distribuciones de $n$ variables usando solo las probabilidades condicional y marginal.

Dada una permutación de números entre 1 y $n$, $\pi = i_1 i_2 ... i_n$, definimos una clase de distribuciones de probabilidad, $\widehat{p}_\pi(X)$:

$$\widehat{p}_\pi(X) = p(X_{i_1}|X_{i_2})p(X_{i_2}|X_{i_3})...p(X_{i_{n-1}}|X_{i_n})p(X_{i_n})$$

La distribución $\widehat{p}_\pi(X)$ usa $\pi$ como un ordenamiento para las parejas de probabilidades condicionales. Nuestra meta es elegir la permutación $\pi$ que maximice el acuerdo entre $\widehat{p}_\pi(X)$ y la distribución $p(X)$. La similitud entre dos distribuciones puede ser medida por la divergencia de Kullback-Liebler:

$$\begin{aligned} D(p||\widehat{p}_\pi) &= \int_\chi p[\log p - \log \widehat{p}_\pi] dX \\ &= E_p[\log p] - E_p[\log \widehat{p}_\pi] \\ &= -h(p) - E_p[\log p(X_{i_1}|X_{i_2})p(X_{i_2}|X_{i_3})...p(X_{i_{n-1}}|X_{i_n})p(X_{i_n})] \\ &= -h(p) + h(X_{i_1}|X_{i_2}) + h(X_{i_2}|X_{i_3}) + ... + h(X_{i_{n-1}}|X_{i_n}) + h(X_{i_n}) \end{aligned}$$

Esta divergencia es siempre no negativa, con igualdad sólo en el caso donde $\widehat{p}(\pi)$ y $p(X)$ son distribuciones idénticas. El $\pi$ óptimo esta definido como el que minimiza esta divergencia. Para una distribución que puede ser completamente descrita por parejas de probabilidades condicionales, el $\pi$ óptimo genera una distribución que sería idéntica a la distribución verdadera. Buscando sobre todas las $n!$ permutaciones, es posible determinar el $\pi$ óptimo. En interés de eficiencia computacional, empleamos un algoritmo glotón directo para escoger una permutación:

1. $i_n = \arg \min_j \widehat{h}(X_j)$
2. $i_k = \arg \min_j \widehat{h}(X_j|X_{i_{k+1}})$, donde
   $j \neq i_{k+1}...i_n$ y $k = n-1, n-2, ..., 2, 1$.

donde $\widehat{h}()$ es la entropia empírica. Una vez que la distribución es elegida, la generación de las siguientes generaciones es directa:

1. Elegir un valor para $X_{i_n}$ basado en su probabilidad empírica $\widehat{p}(X_{i_n})$.
2. Para $k = n-1, n-2, ..., 2, 1$, elegir el elemento $X_{i_k}$ basado en la probabilidad condicional empírica $\widehat{p}(X_{i_k}|X_{i_{k+1}})$.

## 3.4. Algoritmo BMDA

Pelikan y Mühlenbein [16] propusieron una factorización de la distribución de probabilidad que solo necesita estadísticos de segundo orden. Este método está basado en la construcción de un grafo de dependencias, el cual es siempre acíclico pero no necesariamente es un grafo conectado. De hecho, el grafo de dependencias puede ser visto como un conjunto de árboles que no están mutuamente conectados. La idea básica de construcción del grafo de dependencias es simple.

### 3.4.1. Probabilidades Marginales y Estadísticos de Pearson

Denotemos la longitud del cromosoma como $n$. Sea $P$ el conjunto de cadenas binarias de longitud $n$ (la población). El tamaño de $P$ estará denotado por $N$. Para cada posición $i \in \{0, ..., n-1\}$ y cada posible valor en esta posición $x_i \in \{0, 1\}$, definimos la frecuencia univariada marginal $p_i(x_i)$ para el conjunto $P$ como la frecuencia de cadenas que tienen $x_i$ en la $i$-ésima posición en el conjunto $P$. Similarmente para cada dos posiciones $i \neq j \in \{0, ..., n-1\}$ y cualesquiera posibles valores en estas posiciones $x_i, x_j \in \{0, 1\}$ definimos la probabilidad marginal divariada $p_{i,j} = (x_i, x_j)$ para el conjunto $P$ como la frecuencia de cadenas que tienen $x_i$ y $x_j$ en las posiciones $i$ y $j$ respectivamente.

Algunas veces el término probabilidad será usado en lugar de frecuencia. Con el uso de frecuencias marginales divariadas y univariadas, la probabilidad condicional del valor $x_i$ en la $i$-ésima posición dado el valor $x_j$ en la $j$-ésima posición se puede calcular como

$$p_{i,j}(x_i|x_j) = \frac{p_{i,j}(x_i, x_j)}{p_j(x_j)} \tag{3.1}$$

El estudio chi cuadrada de Pearson está definido por:

$$\chi^2 = \sum \frac{(observado - esperado)^2}{esperado} \tag{3.2}$$

Aquí para cada par de posiciones la cantidad observada es el número de pares posibles de valores en esas posiciones. Si estas dos posiciones fueron independientes, el número para cada par de valores podría ser fácilmente calculado usando la teoría básica de probabilidad. Esta es la

cantidad esperada. Entonces, en términos de frecuencias univariadas y bivariadas y el número total de puntos tomados en cuenta, para posiciones $i \neq j$ tenemos:

$$\chi^2_{i,j} = \sum_{x_i,x_j} \frac{(Np_{i,j}(x_i,x_j) - Np_i(x_i)p_j(x_j))^2}{Np_i(x_i)p_j(x_j)} \tag{3.3}$$

Si las posiciones $i$ y $j$ son independientes para el 95 %, entonces el estadístico chi cuadrada de Pearson queda con la siguiente desigualdad:

$$\chi^2_{i,j} < 3{,}84 \tag{3.4}$$

### 3.4.2. Construcción del Grafo de Dependencias

El grafo estará definido por 3 conjuntos, $V$, $E$ y $R$, es decir $G = (V, E, R)$. $V$ es el conjunto de vértices, $E \subset V \times V$ es el conjunto de ejes y $R$ es un conjunto que contiene un vértice para cada componente conectado de $G$. En un grafo de dependencia cada nodo corresponde a una posición en la cadena. Hay una correspondencia uno a uno entre los vértices y las posiciones en una cadena. Para esto podemos usar el conjunto de vértices $V = \{0, ..., n-1\}$ donde el vértice $i$ corresponda a la pésima posición. La generación de nuevas cadenas no depende del número de componentes conectados en el grafo de dependencia.

Denotemos $A$ como el conjunto de vértices que no serán procesados aún. Al principio del algoritmo $A$ es igual a $V$. Entonces sucesivamente, cuando los ejes sean añadidos a $E$, $A$ irá decreciendo. El algoritmo termina cuando $A$ es un conjunto vacío, lo que significa que todos los vértices han sido procesados. Otro conjunto $D$, es el conjunto de pares de $V \times V$ que no son independientes para el $95\,\%$ (Ecuaciones 3.3 y 3.4), es decir:

$$D = \{(i,j) | i \neq j \in \{0, ..., n-1\} \wedge \chi^2_{i,j} \geq 3{,}84\} \tag{3.5}$$

Los pasos necesarios para construir el grafo de dependencia se muestran en el algoritmo 4.

### 3.4.3. Generacion de Nuevos individuos

Para generar los nuevos individuos se utiliza el grafo de dependencias $G = (V, E, R)$. Para cada individuo se generan los valores para

**Algoritmo 4** Algoritmo para la Construcción del Grafo de Dependencias

**Inicializar:** $V \leftarrow \{0, ..., n-1\}$ , $A \leftarrow V$ , $E \leftarrow \emptyset$

1: $v \leftarrow$ cualquier vértice de $A$
   Agregar $v$ a $R$
2: Quitar $v$ de $A$
3: **si** no hay mas vértices en $A$ **entonces**
4: Terminar
5: **fin si**
6: **si** Si no hay mas dependencias en $D$ de cualquier $v$ y $v'$ donde $v \in A$ y $v' \in V \backslash A$ **entonces**
7: Ir al paso 1
8: **fin si**
9: Fijar $v$ como el vértice de $A$ que maximiza $X^2_{v,v'}$, sobre todo $v \in A$ y $v' \in V \backslash A$
10: Agregar el eje $(v, v')$ al conjunto de ejes $E$
11: Ir al paso 2

posiciones contenidas en $R$ por medio de las probabilidades marginales univariadas. Entonces, si existe una posición $v$ que aún no ha sido generada y está conectada a alguna posición $v'$ que ya ha sido generada (de acuerdo con el conjunto de ejes $E$), es generada usando la probabilidad condicional (Ecuacion 3.1 para una posición $v$ dado el valor de la posición $v'$. El ultimo paso se repite hasta que los valores de todas las posiciones han sido generados.

En el algoritmo 5 se describe la generación de un nuevo individuo, aparece un conjunto importante entre los definidos por el grafo $G$, lo denotaremos como $K$ y cuenta por el conjunto de todas las posiciones que ya han sido generadas. El individuo es una cadena de longitud $n$ y será denotada por $x$. Si $i$-ésima posición será denotada como $x_i$.

El conjunto $K$ está inicializado como un conjunto finito y en cada ciclo se quita un vértice de el.Para cada componente conectado, al menos un vértice es generado primero.

La estructura del algoritmo se muestra en la figura 3.2

### 3.4.4. Descripción del Algoritmo

Una vez definidos los algoritmos para la construcción del grafo de dependencias y la generación de nuevos individuos, describiremos el Algo-

**Algoritmo 5** Algoritmo para la Generación de Nuevos Individuos

**Inicializar:** $K \leftarrow V$

1: Generar $x_r$ para toda $r \in R$ usando las probabilidades marginales, por ejemplo, generar el valor $a$ con una probabilidad de $p_r(a)$
   $K \leftarrow K \backslash R$
2: **si** $K$ está vacio **entonces**
3:  Terminar
4: **fin si**
5: Elegir $k$ de $K$ tal que exista un $k'$ de $V \backslash K$ conectado a $k$ en el grafo $G$
6: Generar $x_k$ usando la probabilidad condicional dado el valor de $x_{k'}$, por ejemplo, genera el valor de $a$ con probabilidad $p_{k,k'}(a|x_{k'})$
7: Quitar $k$ del conjunto $K$
8: Ir al paso 5

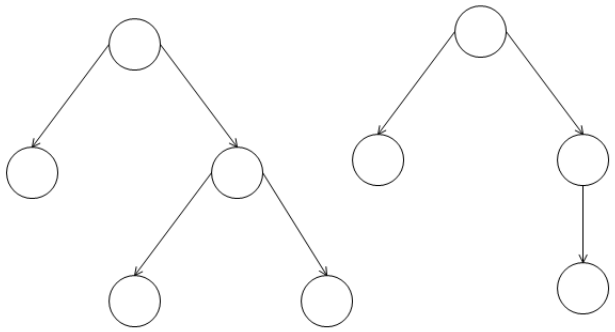


Figura 3.2: Estructura del BMDA

ritmo de Distribución Marginal Bivariada. En este algoritmo la población inicial se genera de manera aleatoria. De esta población, se seleccionan los mejores individuos. Se calculan las probabilidades marginales univarias y bivariadas y haciendo uso de de estas probabilidades, se construye el grafo de dependencias. Tomando en cuenta el grafo de dependencias, se generan nuevos individuos. Los nuevos individuos se agregan a la generación anterior de la cual los individuos fueron seleccionados originalmente. Estos nuevos individuos reemplazan a algunos de los anteriores, generalmente a los peores, por lo que el tamaño de la población permanece constante. Este proceso se repite hasta que un determinado criterio de paro se cumple.

A continuación se muestra el algoritmo del BMDA:

**Algoritmo 6** Algoritmo del BMDA

**Inicializar:** $t \leftarrow 0$
Generar poblacion inicial $P_0$ aleatoriamente
1: Seleccionar una muestra $S(t)$ de $P(t)$)
Calcular las probabilidades marginales $p_i$ y las probabilidades condicionales $p_{i,j}$ para la muestra seleccionada
2: Crear el grafo de dependencias $G = (V, E, R)$ usando las frecuencias $p_i$ y $p_{i,j}$
3: Generar el conjunto de nuevos individuos $O(t)$ usando el grafo de dependencias y las probabilidades $p_i$ y $p_{i,j}$
4: Reemplazar algunos individuos de $P(t)$ con los nuevos individuos $O(t)$
$t \leftarrow t + 1$
5: **si** aún no se cumple el criterio de paro **entonces**
6: Ir al paso 1
7: **fin si**

## 3.5. Árbol de Dependencias de Chow y Liu

El método para aprendizaje estructural de árboles se basa en el algoritmo desarrollado para aproximar una distribución de probabilidad por un producto de probabilidades de segundo orden, lo que corresponde a un árbol [3]. La probabilidad conjunta de $n$ variables se puede representar (aproximar) como se muestra en la ecuación 3.6.

$$p(x) = \prod_{i=1}^{n} p(x_i | x_{j(i)}) \tag{3.6}$$

donde $x_{j(i)}$ es la variable designada como padre de $x_i$ en alguna orientación del árbol.

Se plantea el problema como uno de optimización y lo que se desea es obtener la estructura en forma de árbol que más se aproxime a la distribución real. Para ello se utiliza la divergencia de Kulback-Liebler como una medida de la diferencia de información entre la distribución real $P(x)$ y la aproximada $P_\alpha(x)$, la cual se observa en la ecuación 3.7.

$$I(P, P_\alpha) = \sum_{X} P(X) \log \frac{P(X)}{P_\alpha(X)} \tag{3.7}$$

Esta diferencia la podemos ver tambien en función de dos variables. Supongamos que $X = x, y$, entonces tenemos que $P(X) = P(x, y)$, y bajo el supuesto de que $x$ y $y$ son independientes, entonces tenemos que $P_\alpha(X) = P(x)P(y)$, susutituyendo en la ecuación 3.7 tenemos:

$$I(x, y) = \sum_x \sum_y P(x, y) \log \frac{P(x, y)}{P(x)P(y)} \tag{3.8}$$

Entonces el objetivo es minimizar $I$. Como podemos observar en la ecuación 3.8 tenemos la información mutua entre dos de variables, la cual se deriva de la diferencia de Kulback-Liebler.

Se puede demostrar que la diferencia de información es una función del negativo de la suma de las informaciones mutuas (pesos) de todos los pares de variables que constituyen el árbol. Por lo que encontrar el árbol más próximo equivale a encontrar el árbol con mayor peso. Basado en lo anterior, el procedimiento para determinar árbol Bayesiano óptimo a partir de datos se muestra en el algoritmo 7 [14].

**Algoritmo 7** Algoritmo del Árbol de Dependendencias de Chow y Liu

1: Comenzar con un grafo sin aristas $G$
2: Calcular la información mutua $I(x_i, x_j)$ con $i < j$ donde $i, j = 1, 2, ..., n$ Asignar este valor como peso al arco que conecta a las variables $x_i$ y $x_j$
3: Ordenar $I(x_i, x_j)$ en orden decreciente
4: Considerar un árbol inicial en el que no existen arcos entre las variables
5: Asignar los dos arcos de mayor peso al árbol
6: Examinar el siguiente arco de mayor peso, y añadirla al árbol a no ser que forme un ciclo, en cuyo caso se descarta y se examina el siguiente arco de mayor peso.
7: Repetir el paso 6 hasta seleccionar $n - 1$ arcos.
8: Transformar el árbol no dirigido resultante en uno dirigido, escogiendo una variable como raíz, para a direccionar el resto de arcos.

Entre las ventajas que ofrece este algoritmo están el que para calcular la cantidad de información sólo se utilizan distribuciones conjuntas bidimensionales, las cuales pueden ser calculadas de forma eficiente y fiable a partir de un número no muy elevado de datos y si la distribución es representable por un árbol, el algoritmo recupera el árbol que la representa. La estructura del algoritmo del árbol se muestra en la figura 3.3.

Además cabe mencionar que este algoritmo es la base del *Poliárbol*,

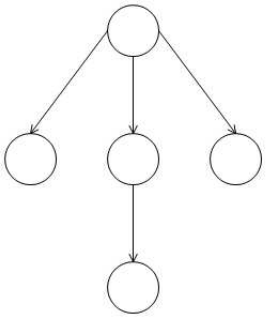


Figura 3.3: Estructura del Árbol de Dependencias de Chow y Liu

el cual es la base de este trabajo de tesis y cuyo algoritmo se describe a detalle en el capítulo 4.

# Capítulo 4

# Algoritmo de Aprendizaje del Poliárbol

## 4.1. Introducción

Este método se puede considerar como una extensión del método de Chow y Liu[3]. El poliárbol es un grafo simplemente conectado es decir, que un nodo puede tener varios padres, pero solo debe existir una trayectoria entre dos nodos. Para efectos de este trabajo de tesis sólo se consideraron dos padres para algún nodo. El algoritmo de propagación es muy similar al de los árboles. La principal diferencia es que se requiere de la probabilidad conjunta de cada nodo dado todos sus padres, por lo que es posible distinguir, dado el subgrafo $x - y - z$, la estructura $x \rightarrow y \leftarrow z$, en donde podemos determinar si $x$ y $z$ son padres de $y$. Otra manera de verlo es buscando qué tanta información pueden darme las variables $x$ y $z$ dada la variable $y$.

Esto lo podemos observar en nuestro entorno, ya que dos eventos que aparentemente son independientes entre si pueden relacionarse por medio de un tercero. Por ejemplo, si queremos encontrar la relación que existe entre un tipo de carretera y un tipo de auto, podemos definir dos variables: $A$ =**Tipo de Auto y** $C$ =**Tipo de Carretera**. Como se muestra en la figura 4.1(a).

Suponiendo que la carretera puede ser una autopista o un camino terracería, y que el auto puede ser una camioneta o una carro antiguo, entonces tratamos de encontrar una relación entre ellas, por ejemplo,

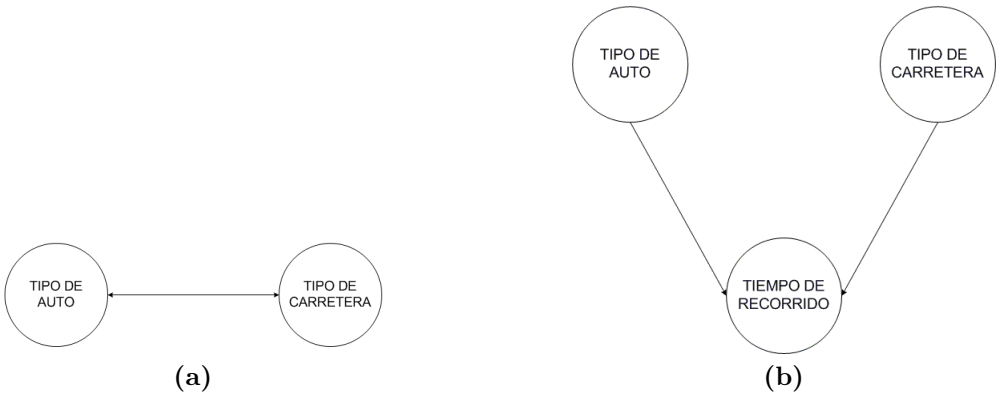



Figura 4.1: Variables de la vida cotidiana

tenemos conocimiento de que la carretera es una autopista, pero eso no nos da información acerca del auto, es decir La variable $C$ no nos dice nada acerca de la variable $A$ y viceversa.

Si a nuestro ejemplo le incluimos una tercer variable $T$ =**Tiempo de recorrido**, como se muestra en la figura 4.1(b) entonces las dos primeras variables nos pueden dar información una de otra, por ejemplo, si sabemos que el auto es una camioneta y que el tiempo de recorrido fue muy largo, entonces podemos decir que la carretera es un camino de terracería.

Para efectos del algoritmo del Poliárbol, centraremos nuestra atención en este tipo de relaciones existentes entre tres variables. En los poliárboles es posible encontrar tres tipos de tripletas adyacentes:

1. Secuencial: $X_i \rightarrow X_k \rightarrow X_j$
2. Divergente: $X_i \leftarrow X_k \rightarrow X_j$
3. Convergente: $X_i \rightarrow X_k \leftarrow X_j$

La tripleta $X_i \rightarrow X_k \leftarrow X_j$ se llama *patrón cabeza-cabeza* o *conexión cabeza-cabeza* y el nodo hacia el cual convergen los arcos se conoce como *nodo cabeza-cabeza*.

Las dos primeras tripletas son probabilísticamente indistinguibles ya que representan las mismas relaciones de dependencia/independencia

entre las variables $X_i$ y $X_j$, ya que son marginalmente dependientes y condicionalmente independientes dado $X_k$ [14]. Sin embargo, la tercera tripleta si puede ser reconocida, debido a que solamente en este tipo $X_i$ y $X_j$ son marginalmente independientes y condicionalmente dependientes dado $X_k$.

1. Verificación de la independencia en el caso Secuencial[9]

$$P(X_i, X_j, X_k) = P(X_i)P(X_j|X_i)P(X_k|X_j) \tag{4.1}$$

en donde

$$P(X_j|X_i) = \frac{P(X_i, X_j)}{P(X_i)} \tag{4.2}$$

Sustituyendo 4.2 en 4.1, tenemos

$$P(X_i, X_j, X_k) = P(X_i)\frac{P(X_i, X_j)}{P(X_i)}P(X_k|X_j) \tag{4.3}$$

Reduciendo algunos terminos algebráicos nos queda la ecuación 4.4

$$P(X_i, X_j, X_k) = P(X_i, X_j)P(X_k|X_j) \tag{4.4}$$

Ahora, tenemos que la probabilidad condicional de $X_k$ dadas las variables $X_i$ y $X_j$ queda expresada en la ecuación 4.5

$$P(X_k|X_i, X_j) = \frac{P(X_i, X_j, X_k)}{P(X_i, X_j)} \tag{4.5}$$

Sustituyendo 4.4 en 4.5 tenemos

$$P(X_k|X_i, X_j) = \frac{P(X_i, X_j)P(X_k|X_j)}{P(X_i, X_j)} \tag{4.6}$$

Por reudcción algebráica nos queda

$$P(X_k|X_i, X_j) = P(X_k|X_j) \tag{4.7}$$

En la ecuación 4.7 podemos observar que no es necesaria la variable $X_i$ para generar la variable $X_k$, por lo que decimos que $X_k$ es condicionalmente independiente de $X_i$.

2. Verificación de la independencia en el caso Divergente

$$P(X_i, X_j, X_k) = P(X_j)P(X_k|X_j)P(X_i|X_j) \tag{4.8}$$

en donde

$$P(X_i|X_j) = \frac{P(X_i, X_j)}{P(X_j)} \tag{4.9}$$

Sustituyendo 4.9 en 4.8, tenemos

$$P(X_i, X_j, X_k) = P(X_j)\frac{P(X_i, X_j)}{P(X_j)}P(X_k|X_j) \tag{4.10}$$

Reduciendo algunos terminos algebráicos nos queda la ecuación 4.11

$$P(X_i, X_j, X_k) = P(X_i, X_j)P(X_k|X_j) \tag{4.11}$$

Ahora, tenemos que la probabilidad condicional de $X_k$ dadas las variables $X_i$ y $X_j$ queda expresada en la ecuación 4.12

$$P(X_k|X_i, X_j) = \frac{P(X_i, X_j, X_k)}{P(X_i, X_j)} \tag{4.12}$$

Sustituyendo 4.11 en 4.12 tenemos

$$P(X_k|X_i, X_j) = \frac{P(X_i, X_j)P(X_k|X_j)}{P(X_i, X_j)} \tag{4.13}$$

Por reudcción algebráica nos queda

$$P(X_k|X_i, X_j) = P(X_k|X_j) \tag{4.14}$$

En la ecuación 4.14 podemos observar que no es necesaria la variable $X_i$ para generar la variable $X_k$, por lo que decimos que $X_k$ es condicionalmente independiente de $X_i$.

3. Verificación de la independencia en el caso Convergente

$$P(X_i, X_j, X_k) = P(X_i)P(X_j)P(X_k|X_i, X_j) \tag{4.15}$$

en donde

$$P(X_k|X_i, X_j) = \frac{P(X_i, X_j, X_k)}{P(X_i, X_j)} \tag{4.16}$$

Sustituyendo 4.16 en 4.15, tenemos

$$P(X_i, X_j, X_k) = P(X_i)P(X_j)\frac{P(X_i, X_j, X_k)}{P(X_i, X_j)} \tag{4.17}$$

Y como $X_i$ y $X_j$ son independientes, entonces tenemos que el producto de sus probabilidades marginales es igual a su probabilidad conjunta, como se muestra en 4.18

$$P(X_i, X_j) = P(X_i)P(X_j) \tag{4.18}$$

Sustiuyendo esto en 4.17, y reduciendo algunos terminos algebráicos nos queda la ecuación 4.19

$$P(X_i, X_j, X_k) = P(X_i, X_j, X_k) \tag{4.19}$$

La demostración de la dependencia condicional para el caso convergente está desarrollada en el libro de Finn V. Jensen [9].

Como podemos ver las ecuaciones 4.4 y 4.11 son iguales, por esto decimos que el caso secuencial y el divergente no son distinguibles probabilisticamente.

Ahora que hemos visto la razón por la cual la estructura convergente es importante para el poliárbol, seguiremos mencionando los detalles importantes de éste. Hay que tomar en cuenta que el número máximo de aristas que se pueden insertar es $n-1$, restricción que garantiza que la estructura resultante sea simplemente conectada.

Para la inserción de las aristas nos basamos en dos medidas: *la información mutua marginal y la información mutua condicional*, los cuales son valores no negativos[15].

La información mutua marginal $I(X_i, X_j)$ de las variables aleatorias $X_i$ y $X_j$ se define como

$$I(X_i, X_j) = \sum_{x_i, x_j} p(x_i, x_j) \log_2 \frac{p(x_i, x_j)}{p(x_i)p(x_j)} \geq 0 \tag{4.20}$$

La información mutua condicional $I(X_i, X_j|X_k)$ de las variables aleatorias $X_i$ y $X_j$ dado $X_k$ se define como

$$I(X_i, X_j|X_k) = \sum_{x_i,x_j,x_k} p(x_i, x_j, x_k) \log_2 \frac{p(x_i, x_j, x_k)p(x_k)}{p(x_i, x_k)p(x_j, x_k)} \geq 0 \tag{4.21}$$

La única diferencia con el algoritmo propuesto por Chow y Liu en la inserción de las aristas radica en la ecuación 4.21, ya que, mientras Chow y Liu [3] utilizan solo la información mutua marginal entre dos variables, la cual también queda expresada como la dependencia marginal $Dep(X_i, X_j)$ existente entre dichas variables, el poliárbol utiliza otra medida conocida como dependencia global $Dep_g(X_i, X_j)$, la cual se define como

$$Dep_g(X_i, X_j) = \min_{\{X_k\}}(Dep(X_i, X_j), I(X_i, X_j|X_k)) \tag{4.22}$$

Solo se insertan aquellas aristas cuyos valores $Dep(X_i, X_j)$ y $Dep_g(X_i, X_j)$ superan dos pesos umbrales (pequeños valores reales positivos $\epsilon_0$ y $\epsilon_1$, dados como paramétros del algoritmo), que en este caso ambos tienen valores de $0{,}02$ y que además no violan las restricciones correspondientes el modelo en cuestión. Por ejemplo, ninguna arista que cree un ciclo puede insertarse.

Para ilustrar el comportamiento de las ecuaciones 4.20 y 4.21 se realizón sencillo experimento con tres variables aleatorias, cuyos valores podemos ver en el cuadro 4.1.

Utilizando la ecuación 4.20, checamos la dependencia existente entre las variables $X_1$ y $X_2$, esto con la finalidad de saber qué tanto me puede decir una variable de la otra, los resultados de los parámetros utilizados para medir dicha dependencia se muestran en el cuadro 4.2.

Todos los valores del cuadro 4.2 se utilizaron para calcular el valor de dependencia condicional usando la ecuación 4.20, dicho valor es $I(X_1, X_2) = 0{,}07156$.

Este valor lo que nos indica es que existe una pequeña dependencia entre las variables $X_1$ y $X_2$, la cual no es muy significativa dado su valor tan pequeño pero dado el umbral que se uso para definir si existe dependencia o no entre las variables, podemos decir que si existe.

| $X_1$ | $X_2$ | $X_3$ |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |

Cuadro 4.1: Valores de las variables aleatorias $X_1$, $X_2$ y $X_3$

Ahora, utilizamos una tercer variable $X_3$ y usando la ecuación 4.21, medimos la dependencia que existe entre $X_1$ y $X_2$ dado $X_3$, los resultados de los parámetros utilizados para medir dicha dependencia se muestran en el cuadro 4.3 y dichos valores se utilizaron para calcular la información condicional que existe entre $X_1$ y $X_2$ dada la tercer variable $X_3$, dicho valor $I(X_1, X_2|X_3) = 0{,}1422$ nos indica que existe un valor de dependencia mas fuerte cuando utilizamos una tercer variable, es decir, dado el concepto que utilizamos anteriormente, podemos decir que las variables $X_1$ y $X_2$ nos dan más información una de la otra por medio de una tercer variable $X_3$ que por si solas. En este caso las variables $X_1 - X_3 - X_2$ forman un *patrón cabeza-cabeza* del poliárbol.

## 4.2. Algoritmo de Aprendizaje del Poliárbol

Ya que hemos visto las bases del poliárbol, a continuación mostraremos la serie de pasos que nos van a servir para formar dicha estructura,

| | |
|---|---|
| $P(X_1 = 1) = 0{,}466667$ | $P(X_2 = 1) = 0{,}8$ |
| $P(X_1 = 1, X_2 = 1) = 0{,}3$ | $P(X_1 = 1, X_2 = 0) = 0{,}1667$ |
| $P(X_1 = 0, X_2 = 1) = 0{,}5$ | $P(X_1 = 0, X_2 = 0) = 0{,}0333$ |
| $P(X_2 = 0 \mid X_1 = 0) = 0{,}0625$ | $P(X_2 = 0 \mid X_1 = 1) = 0{,}3571$ |
| $P(X_2 = 1 \mid X_1 = 0) = 0{,}9375$ | $P(X_2 = 1 \mid X_1 = 1) = 0{,}6428$ |

Cuadro 4.2: Valores utilizados para medir dependencia entre $X_1$ y $X_2$

| | |
|---|---|
| $P(X_1 = 1) = 0{,}466667$ | $P(X_2 = 1) = P(X_3 = 1) = 0{,}8$ |
| $P(X_1 = 1, X_2 = 1) = 0{,}3$ | $P(X_1 = 1, X_2 = 0) = 0{,}1667$ |
| $P(X_1 = 0, X_2 = 1) = 0{,}5$ | $P(X_1 = 0, X_2 = 0) = 0{,}0333$ |
| $P(X_1 = 1, X_3 = 1) = 0{,}3333$ | $P(X_1 = 1, X_3 = 0) = 0{,}1333$ |
| $P(X_1 = 0, X_3 = 1) = 0{,}4667$ | $P(X_1 = 0, X_3 = 0) = 0{,}0667$ |
| $P(X_2 = 1, X_3 = 1) = 0{,}6667$ | $P(X_2 = 1, X_3 = 0) = 0{,}1333$ |
| $P(X_2 = 0, X_3 = 1) = 0{,}1333$ | $P(X_2 = 0, X_3 = 0) = 0{,}0667$ |
| $P(X_1 = 0, X_2 = 0, X_3 = 0) = 0{,}0333$ | $P(X_1 = 0, X_2 = 0, X_3 = 1) = 0$ |
| $P(X_1 = 0, X_2 = 1, X_3 = 0) = 0{,}0333$ | $P(X_1 = 0, X_2 = 1, X_3 = 1) = 0{,}4667$ |
| $P(X_1 = 1, X_2 = 0, X_3 = 0) = 0{,}0333$ | $P(X_1 = 1, X_2 = 0, X_3 = 1) = 0{,}1333$ |
| $P(X_1 = 1, X_2 = 1, X_3 = 0) = 0{,}1$ | $P(X_1 = 1, X_2 = 1, X_3 = 1) = 0{,}2$ |
| $P(X_3 = 0 \mid X_1 = 0, X_2 = 0) = 1$ | $P(X_3 = 0 \mid X_1 = 0, X_2 = 1) = 0{,}0667$ |
| $P(X_3 = 0 \mid X_1 = 1, X_2 = 0) = 0{,}1999$ | $P(X_3 = 0 \mid X_1 = 1, X_2 = 1) = 0{,}3333$ |
| $P(X_3 = 1 \mid X_1 = 0, X_2 = 0) = 0$ | $P(X_3 = 1 \mid X_1 = 0, X_2 = 1) = 0{,}9334$ |
| $P(X_3 = 1 \mid X_1 = 1, X_2 = 0) = 0{,}7996$ | $P(X_3 = 1 \mid X_1 = 1, X_2 = 1) = 0{,}6667$ |

Cuadro 4.3: Valores utilizados para medir dependencia entre $X_1$ y $X_2$

además de dar una explicación detallada a cada paso, esto con la finalidad de tener una mayor comprensión del poliárbol.

Como podemos ver en el algoritmo 8[15], el poliárbol comienza con un grafo sin aristas, es decir, un conjunto de variables que no tienen ninguna relación entre sí. Tal como se muestra en la figura 4.2. También inicia con una lista vacia $L$. Partiendo de esto, empezamos a buscar dependencias entre las variables, para esto primeramente las tomamos en pares y utilizando la ecuación 4.20, medimos la información mutua marginal de éstas, y si este valor es mayor al umbral $\epsilon_0$, entonces agregamos dichas variables a la lista $L$.

Una vez que tenemos elementos dentro de $L$, se procede a medir la información mutua condicional de cada par de variables contenidas en esta lista con el resto de las variables, para esto utilizamos la ecuación

**Algoritmo 8** Algoritmo del Poliárbol

1: Comenzar con un grafo sin aristas $G$ y una lista vacia $L$
2: **para** cada par de nodos $X_i$ y $X_j$ **hacer**
3: Calcular $Dep(X_i, X_j) = I(X_i, X_j)$
4: **si** $Dep(X_i, X_j) > \epsilon_0$ **entonces**
5: Insertar arco $< X_i, X_j >$ en $L$
6: **fin si**
7: **fin para**
8: **para** cada $< X_i, X_j > \in L$ **hacer**
9: **para** cada nodo $X_k$ donde $k \neq i, j$ **hacer**
10: Calcular $I(X_i, X_j|X_k)$
11: **si** $\exists X_k | I(X_i, X_j|X_k) \leq \epsilon_1$ **entonces**
12: Remover el arco $< X_i, X_j >$ de $L$
13: Seleccionar el próximo arco de $L$
14: **fin si**
15: **fin para**
16: $Dep_g(X_i, X_j) = \text{mín}_{\{X_k\}}(Dep(X_i, X_j), I(X_i, X_j|X_k))$
17: **fin para**
18: Ordenar $L$ en orden decreciente según $Dep_g(X_i, X_j)$
19: Seleccionar arcos de $L$ según el orden e insertar cada arco $< X_i, X_j >$ en $G$, siempre y cuando no genere un ciclo
20: **para** cada subgrafo $X_i - X_k - X_j$ **hacer**
21: **si** $I(X_i, X_j|X_k) > I(X_i, X_j)$ **entonces**
22: Crear patrón cabeza-cabeza $X_i \rightarrow X_k \leftarrow X_j$
23: **fin si**
24: **fin para**
25: Ordenar arcos restantes arbitrariamente

4.21. Si existe una variable cuya medida de información con la pareja de la lista nos da un valor menor o igual al umbral $\epsilon_1$, entonces dicha pareja es removida de la lista $L$. Esto lo hacemos con la finalidad de conservar dentro de la lista solo aquellas variables que tienen dependencias fuertes entre sí o con una tercera variable.

Una vez que hemos depurado la lista $L$, calculamos la dependencia global para cada pareja de esta lista. Para calcular esta dependencia, primeramente se toma el valor menor obtenido de medir la información mutua condicional de este par de variables con respecto al resto de las variables del grafo, una vez que tenemos el valor menor, lo comparamos con el valor de la información mutua marginal existente entre el par de variables de la lista, y tomamos el menor. Y este valor es el que asignamos a la dependencia global. En base a algunos experimentos hechos

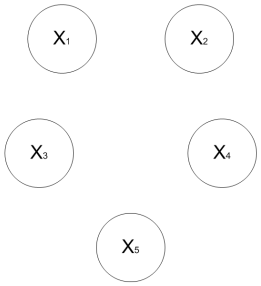



Figura 4.2: Grafo no conectado

para efectos de este trabajo de tesis, se puede ver que el tomar el valor menor o el valor mayor no afecta el desempeño de este algoritmo.

Una vez que tenemos los valores de la dependencia global para cada una de las parejas pertenecientes a la lista $L$, la ordenamos de manera decreciente con respecto a dicha dependencia. Una vez ordenados, comenzamos a insertar los arcos que se encuentran en la lista, excepto que genere un ciclo. Este procedimiento lo hacemos con la finalidad de ir insertando aquellos arcos que unan variables que tengan una fuerte dependencia entre ellas. Y así vamos insertando primero las dependencias más fuertes y dejando al final las más débiles. Un ejemplo de como quedaría el grafo inicial, se muestra en la figura 4.3.

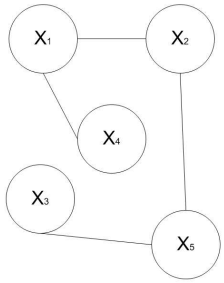



Figura 4.3: Grafo conectado no dirigido

Una vez que hemos insertado las aristas y tenemos un grafo conexo no dirigido, el siguiente paso es orientar los arcos. Primeramente orientamos los patrones cabeza-cabeza. En una conexión $X_i \rightarrow X_k \leftarrow X_j$, conocer el valor de $X_k$ normalmente aumenta el grado de dependencia entre $X_i$ y $X_j$, mientras que en un patrón que no lo es, como es el

caso de las tripletas secuencial y divergente, conocer el nodo del medio produce un efecto contrario, o sea, un decrecimiento en la dependencia entre $X_i$ y $X_j$. Por tanto, para cada conexión $X_i - X_k - X_j$ si se cumple que $I(X_i, X_j|X_k) > I(X_i, X_j)$, entonces creamos el patrón cabeza-cabeza $X_i \rightarrow X_k \leftarrow X_j$.

Para lo anterior, se siguió el siguiente procedimiento. Se identifican todos los posibles patrones cabeza-cabeza y se ordenan de forma decreciente según valor de dependencia global. Posteriormente se procede a orientar los patrones cabeza-cabeza con mayor dependencia global, por ejemplo, si tenemos las tripletas $X_i - X_k - Xj$ y $X_k - X_j - X_m$ y la primer tripleta tiene un valor de dependencia más fuerte, entonces creamos el patrón cabeza-cabeza $X_i \rightarrow X_k \leftarrow X_j$, y el arco existente entre $X_j$ y $X_m$ se orienta arbitrariamente en cualquier dirección, siempre y cuando no genera un cabeza-cabeza falso, si estoy llegase a ocurrir, entonces quitamos esta arista.

## 4.3. Comparación del Algoritmo Genético y el Poliárbol

Como se mencionó en el capítulo 3, el algoritmo genético tiene dependencias entre sus genes las cuales no son medidas ni utilizadas como en el caso de los EDA's.

Para observar estas dependencias se realizó un experimento, el cual consiste en medir las dependencias existentes entre las variables de un EDA (poliárbol) y un AG. Para esto se hicieron corridas con la misma población inicial en ambos algoritmos y conforme se iban generando las nuevas poblaciones se iban midiendo y contando el número de dependencias que había entre las variables de las poblaciones generadas.

El motivo de este experimento es observar el comportamiento de las variables (genes) del AG y hacer un estudio sobre la *epístasis* que pueden llegar a presentar entre ellos. La *epístasis* [17] es la interacción entre diferentes genes en un cromosoma. Si se hace un pequeño cambio a un gen, se espera un cambio resultante en la aptitud del individuo, que puede variar de acuerdo a los valores de otros genes. En términos drásticos, un cambio particular en un gen produce un cambio en la apti-

tud que varía en signo y magnitud. Según Holland[2], es preferible una baja epístasis, pues de esta manera el espacio de búsqueda presenta mayor uniformidad y es fácil obtener buenos bloques constructores.

Lo que se quiere demostrar es que el AG tiene ciertas dependencias entre sus genes que no fueron medidas ni generadas por el algoritmo como lo es en el caso del EDA, en el cual a los datos se les calcula una distribución y en base a ella se generan las nuevas poblaciones.

Para esto se utilizaron dos funciones:

**Onemax**

$$Onemax(x) = \sum_{i=1}^{n} x_i$$

**Deceptive3**

$$Deceptive3(x) = \sum_{i=1}^{l} f_{dec3}(x_{3i-2}, x_{3i-1}, x_{3i})$$

donde $n = 3l$ y $l$ es el número de subfunciones, $u$ es el número de unos en la cadena. Los valores de $u$ y $f_{dec3}$ se muestran en el cuadro 4.4.

| $u$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $f_{dec3}$ | 0.9 | 0.8 | 0 | 1 |

Cuadro 4.4: Valores de la función deceptiva

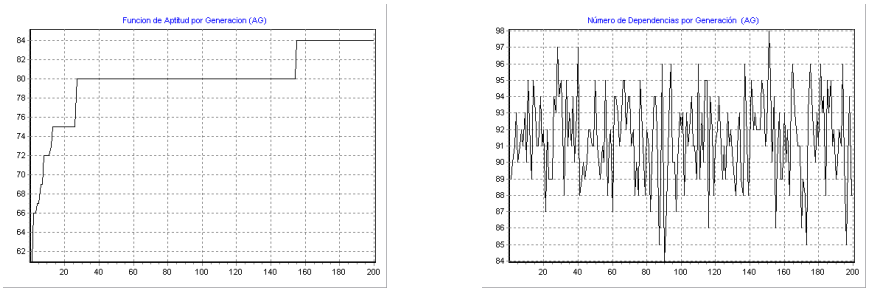
Se utilizaron las ecuaciones 4.20 y 4.21 para medir las dependencias entre las variables de ambos algoritmos. Primeramente calculamos la información mutua marginal entre dos variables, y si este valor es mayor que un umbral (en este caso, el mismo usado en el algoritmo del poliárbol de $0{,}02$), entonces calculamos la información mutua condicional.

A continuación se muestran los resultados de las corridas hechas para resolver ambas funciones usando los dos algoritmos. Cabe mencionar que la funcion $Deceptive3$ pertenece a la familia de las funciones deceptivas, de las cuales se tiene conocimiento de que un genético tiene

problemas para resolverlas, mientras que un EDA las resuelve sin dificultad.

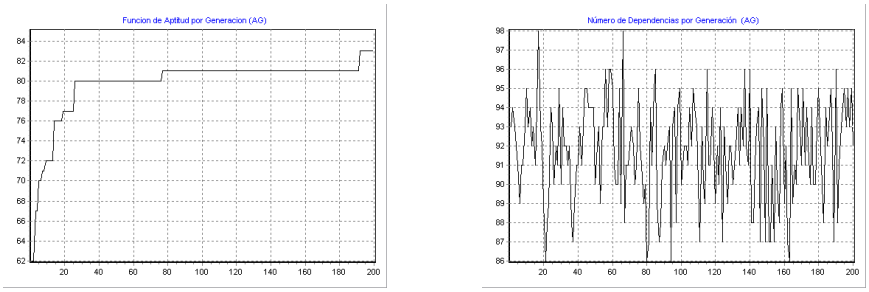
**Onemax**

Se hicieron varias corridas para esta función con los siguientes parámetros: 100 individuos, 200 generaciones, 100 variables. Los resultados se muestran a continuación. Se mostrarán las gráficas de las mejores corridas para cada algoritmo.




Función de Aptitud por Generación. Número de Dependencias por Generación

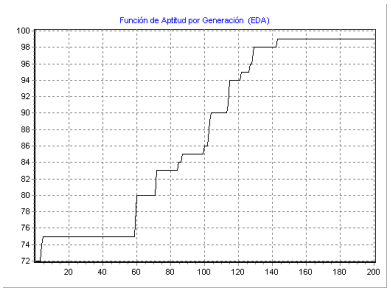
Figura 4.4: Resultados de la Corrida 1 del Algoritmo Genético



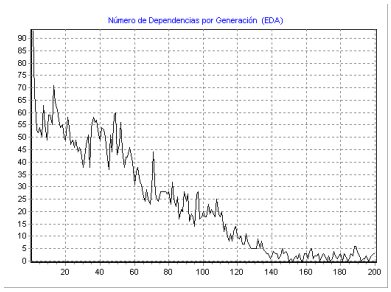

Función de Aptitud por Generación. Número de Dependencias por Generación

Figura 4.5: Resultados de la Corrida 5 del Algoritmo Genético

Como podemos observar en las figuras 4.4 y 4.5 existe una fuerte dependencia entre las variables de las poblaciones generadas por el AG, y dicha dependencia se conserva aun con el paso de las generaciones, caso

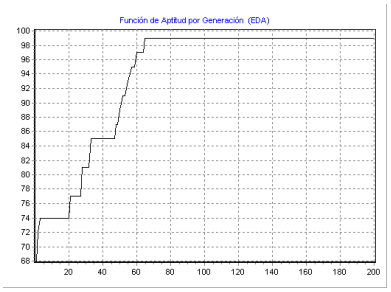



Función de Aptitud por Generación.

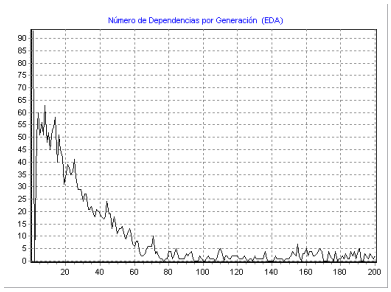



Número de Dependencias por Generación

Figura 4.6: Resultados de la Corrida 3 del Poliárbol




Función de Aptitud por Generación.




Número de Dependencias por Generación

Figura 4.7: Resultados de la Corrida 9 del Poliárbol

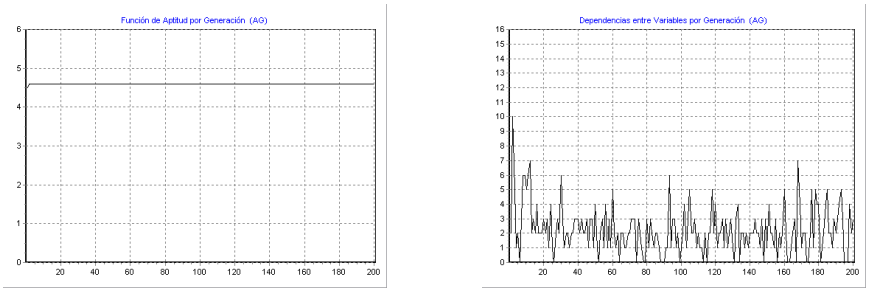
contrario al poliárbol, figuras 4.6 y 4.7, en el cual dichas dependencias se van minimizando conforme se van generando las nuevas poblaciones. Así mismo se puede observar gráficamente que el poliárbol tuvo un mejor desempeño que el AG.

**Deceptive3**

Para esta función se hicieron corridas con la misma población inicial en ambos algoritmos y conforme se iban generando las nuevas poblaciones se iban midiendo y contando el número de dependencias que había entre las variables de las poblaciones generadas.

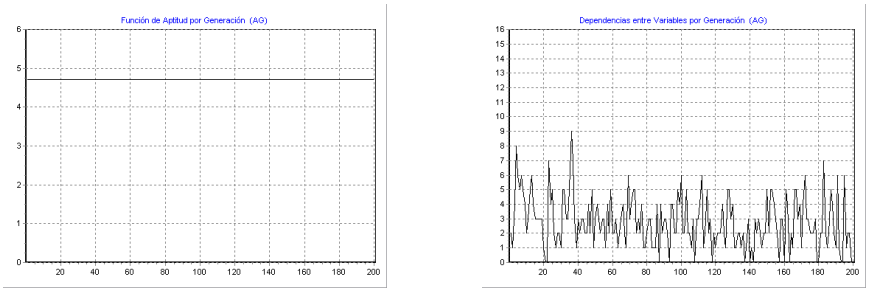
Se hicieron varias corridas para esta función con los siguientes parámetros: 150 individuos, 200 generaciones, 15 variables. Para este caso en particular el óptimo de la función tiene un valor de 5, dada la forma de

la función.




Función de Aptitud por Generación. Número de Dependencias por Generación

Figura 4.8: Resultados de la Corrida 5 del Algoritmo Genético



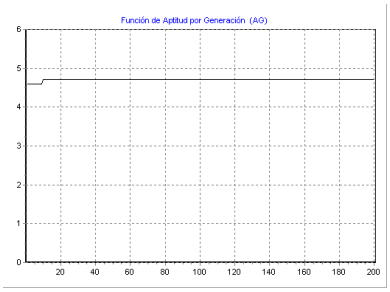

Función de Aptitud por Generación. Número de Dependencias por Generación

Figura 4.9: Resultados de la Corrida 11 del Algoritmo Genético

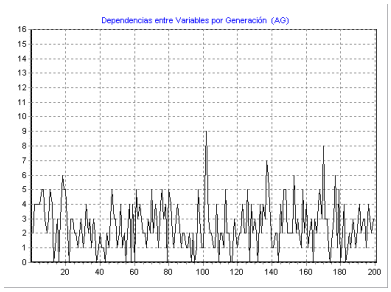
Primeramente veremos el desempeño de ambos algoritmos, como se ve en las figuras 4.8a, 4.9a y 4.10a, el algoritmo genético no llega al óptimo de la función, caso contrario al poliárbol, el cual encuentra el valor óptimo como se ve en las figuras 4.11a, 4.12a y 4.13a.

Como podemos observar en las figuras 4.8b, 4.9b y 4.10b existe en el caso del poliárbol un cambio en la tendencia de las dependencias en las primeras generaciones, esto debido a la inclusión de una mutación que se le implementó al algoritmo para aumentar la diversidad de la población y evitar que se estanque en un mínimo local.

Para ambos algoritmos podemos notar que mientras que el algoritmo

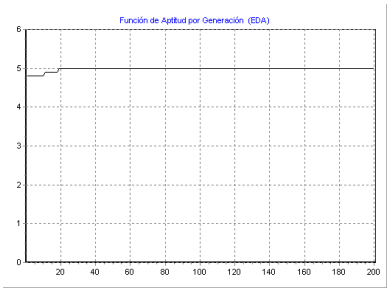



Función de Aptitud por Generación.

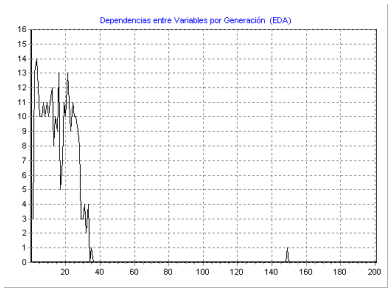



Número de Dependencias por Generación

Figura 4.10: Resultados de la Corrida 21 del Algoritmo Genético




Función de Aptitud por Generación.



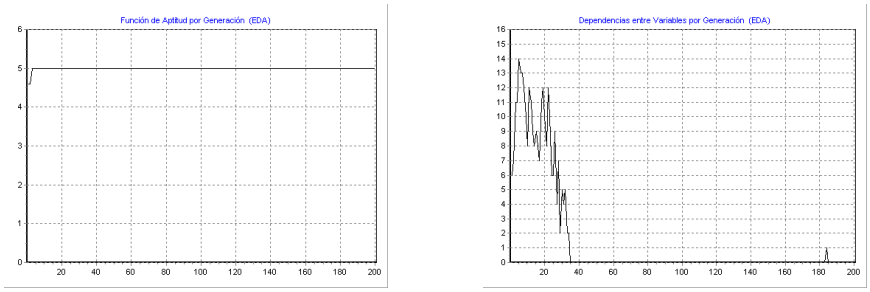

Número de Dependencias por Generación

Figura 4.11: Resultados de la Corrida 3 del Poliárbol

genético tiene varias de sus variables dependientes entre ellas, como se puede ver en las figuras 4.11b, 4.12b y 4.13b, el poliárbol va perdiendo dichas dependencias conforma avanzan las generaciones, y con esto se puede observar un poco el fenómeno de la epístasis en el caso del algoritmo genético.

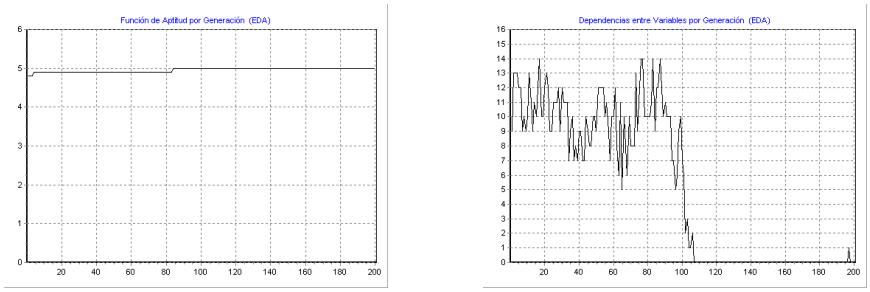
## 4.4. Aplicación del Poliárbol en Hardware Evolutivo

El problema consiste en diseñar un circuito que desempeñe una función lógica deseada (especificada por una tabla de verdad) usando el menor número de compuertas.




Función de Aptitud por Generación. Número de Dependencias por Generación

Figura 4.12: Resultados de la Corrida 9 del Poliárbol




Función de Aptitud por Generación. Número de Dependencias por Generación

Figura 4.13: Resultados de la Corrida 25 del Poliárbol

Para esto se han utilizado las técnicas evolutivas tales como el Algoritmo Genético y el PSO, por mencionar algunas. La propuesta de este trabajo es tratar de resolver el problema planteado anteriormente por medio de Algoritmos de Estimación de Distribución, más específicamente el Poliárbol, ya que se busca aprovechar las dependencias entre las variables y evitar el fenómeno de la epístasis que se presenta en algunas de las técnicas evolutivas.

### 4.4.1. Definición de Parámetros

Una de las ideas iniciales para resolver este problema es representar el circuito dentro de una matriz de tamaño $n \times m$[1], figura 4.14, donde cada elemento de la matriz es una tripleta, que indica el *Tipo de Compuerta, Entrada 1 y Entrada 2*, como se muestra en la figura 4.15.

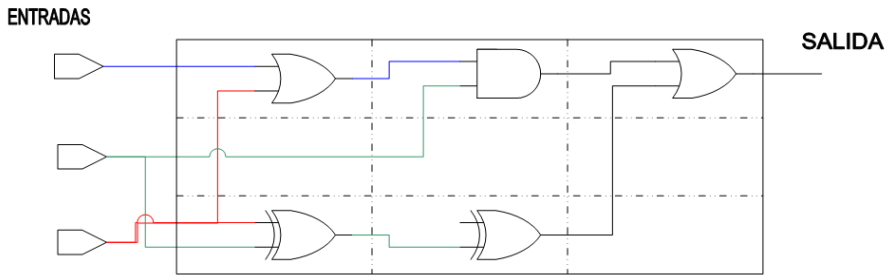



Figura 4.14: Representación del circuito en una matriz

| | | | | | |
|---|---|---|---|---|---|
| Tipo de Compuerta | Entrada 1 | Entrada 2 | Tipo de Compuerta | Entrada 1 | Entrada 2 |
| Tipo de Compuerta | Entrada 1 | Entrada 2 | Tipo de Compuerta | Entrada 1 | Entrada 2 |
| Tipo de Compuerta | Entrada 1 | Entrada 2 | Tipo de Compuerta | Entrada 1 | Entrada 2 |

Figura 4.15: Composición de la matriz

Cada elemento de la tripleta tiene una representación binaria como se puede observar en la figura 4.16.

| Tipo de Compuerta | Entrada 1 | Entrada 2 |
|---|---|---|
| 00 | 01 | 11 |

Figura 4.16: Representación binaria de los elementos de la tripleta

Entonces la representación de la matriz queda ilustrada en la figura 4.17.

Partiendo de la matriz de la figura 4.17 formamos la cadena binaria que se muestra en la figura 4.18. Esta cadena representa un individuo de la población, la cual es pasada como parámetro al EDA, en este caso, poliárbol, para resolver el problema planteado al inicio de esta sección.

Como vimos anteriormente, cada tripleta de unos y ceros representa ya sea el tipo de compuerta, la entrada 1 o la entrada 2 del circuito,

| | | |
|---|---|---|
| 000111 | 011100 | 101111 |
| 110010 | 010100 | 001110 |
| 101010 | 010101 | 110001 |

Figura 4.17: Representación binaria de la matriz

000111011100101111111001001010000111010101001010111 0001

| | | |
|---|---|---|
| 000111 | 011100 | 101111 |
| 110010 | 010100 | 001110 |
| 101010 | 010101 | 110001 |

Figura 4.18: Cadena binaria que representa un individuo

los valores binarios correspondientes al tipo de compuerta se pueden observar en la figura 4.19. Para asignar los valores restantes se aplica modulo a su valor real con el número de compuertas. De esta manera, favorecemos a las primeras tres compuertas.

Una vez que hemos visto la representación que se usará para los individuos de la población, cabe mencionar que el *algoritmo del poliárbol* se aplica por celdas, y no a todo el individuo, es decir, cada nueve bits se ejecuta un poliárbol. Ésto debido a que se observó en las diferentes pruebas realizadas que hacerlo de esta manera arroja mejores resultados.

Una de las razones por las cuales funciona mejor de esta manera es por la información que contiene cada celda, es decir, solo nos da la informacion de la compuerta y sus entradas, es por eso que se considera que las celdas son independientes, si, por ejemplo, la celda tambien tuviera información de a donde va la salida de la compuerta, es decir a que renglon de la siguiente columna debe entrar el valor resultante de la evaluación de ésta, entonces tal vez el considerar independientes las celdas no de buenos resultados, y el aplicar el poliárbol a todo el individuo sea la opción mas viable.

| Tipo de Compuerta | Número que le corresponde |
|---|---|
| AND | 0 (000) |
| NOT | 1 (001) |
| XOR | 2 (010) |
| WIRE | 3 (011) |
| OR | 4 (100) |

Figura 4.19: Valores binarios correspondientes a cada compuerta

### 4.4.2. Caso de Prueba

Como se ha mencionado a lo largo de este capítulo, centramos nuestra atención en las dependencias existentes entre las variables de las poblaciones generadas para ser procesadas por los algoritmos, a continuación se muestran dichas dependencias correspondientes a las variables de las celdas de la matriz que se utilizó para resolver la función lógica que se muestra en el cuadro 4.5[4].

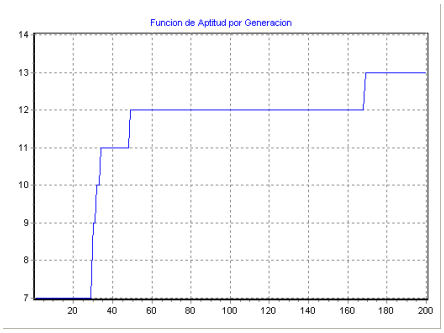
| $A$ | $B$ | $C$ | $S$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Cuadro 4.5: Tabla de Verdad de la Función Lógica utilizada como Caso de Prueba

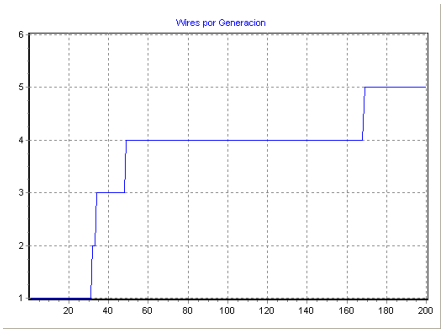
La función de aptitud para este tipo de problemas corresponde al número de coincidencias del circuito representado por el individuo con las salidas de la tabla de verdad una vez que se evaluaron todas sus entradas. Una vez que la función de aptitud corresponde a dicho valor, contamos el número de compuertas *wires* que haya en la matriz y se lo sumamos al valor que ya teníamos. De esta manera aseguramos el menor número de compuertas para resolver cualquiera de las funciones lógicas.

Para este caso de prueba se trabajó con una matriz de $3 \times 3$. Se hicieron 10 corridas del algoritmo, en las cuales la función de aptitud fue de 13, y dichas corridas se hicieron con 200 individuos y 200 generaciones.

El comportamiento de la función de aptitud y de los wires se muestra en las gráficas de la figura 4.20. Como podemos observar, los cambios en la función de aptitud corresponden al incremento de wires.




a




b

Figura 4.20: (a) Función de Aptitud por Generación (b)Número de Wires por Generación

A continuación se presenta un análisis del comportamiento de las dependencias entre las variables que constituyen cada celda de la matriz que se uso para resolver este problema. Para ésto se mostrarán gráficas del número de dependencias existentes en cada celda para cada generación, además de mostrar el componente y las entradas existentes a los que la secuencia de unos y ceros será decodificada.

En la figura 4.21 se muestra la simbología correspondiente para la interpretación de las gráficas.

En la figura 4.22 se muestra el comportamiento de las variables de la celda 1 de la matriz. En dicha celda se observa una dependencia con un alto grado de variabilidad lo cual se ve reflejado en los componentes graficados. Se nota como en las primeras 30 generaciones esta celda contenía una compuerta XOR, lo cual es interpretable a través de los valores obtenidos y a lo indicado en la figura 4.19, posteriormente una

| Color de línea | Significado |
|---|---|
| | No. de dependencias por generación |
| | Tipo de compuerta |
| | Entrada 1 |
| | Entrada 2 |

Figura 4.21: Tabla de Simbología

NOT y finalmente una AND, cambiando además sus entradas.

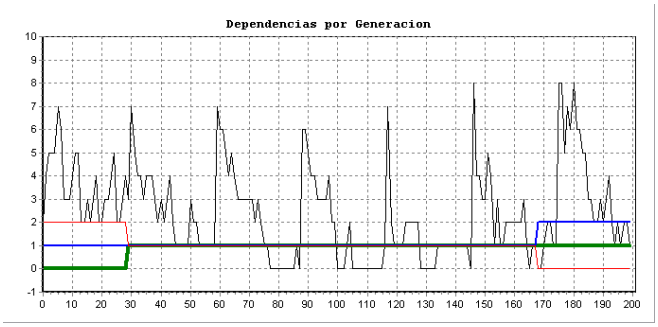



Figura 4.22: Dependencias y valores de la Celda 1

En la celda 2 podemos observar un comportamiento muy uniforme en las dependencias y en las entradas de la compuerta que contiene esta celda, como se muestra en la figura 4.23, observando un cambio favorable al circuito al cambiar una compuerta XOR por un WIRE en las últimas generaciones, lo cual al compararlo con la gráfica del comportamiento de la función de aptitud que se muestra en la figura 4.20a coincide con la generación en la cual se alcanzó el óptimo.

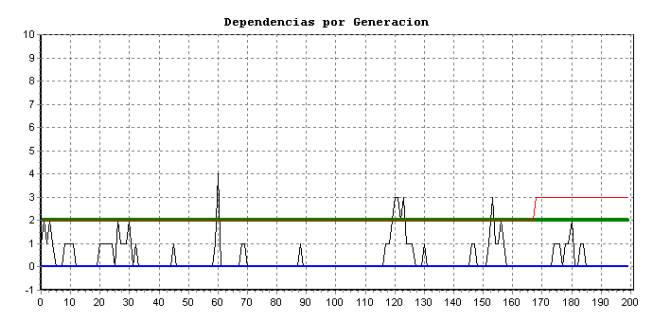



Figura 4.23: Dependencias y valores de la Celda 2

En la figura 4.24 podemos observar el comportamiento de la celda 3, la cual encuentra un valor para la compuerta que favorece al problema que se está resolviendo, y solo presenta un pequeño cambio en una de sus entradas.

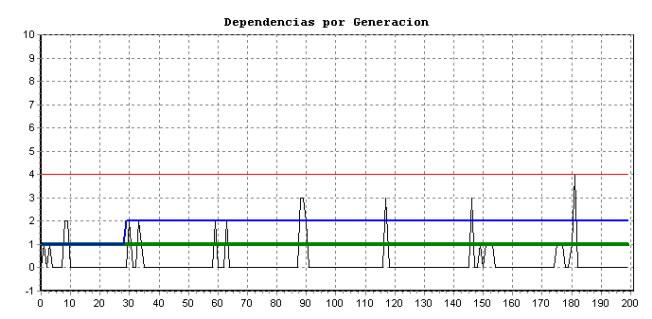



Figura 4.24: Dependencias y valores de la Celda 3

Estas son las celdas correspondientes a la primera columna de la matriz, las cuales reciben sus valores directamente de la tabla de verdad, en este caso las entradas A, B y C están representadas por los valores 0, 1 y 2 respectivamente.

En la figura 4.25 se muestran las dependencias de la celda 4, en esta celda podemos observar nuevamente un comportamiento muy uniforme tanto en el número de dependencias como en la compuerta contenida por dicha celda, mostrando una pequeña variación en una de las entradas de la compuerta.

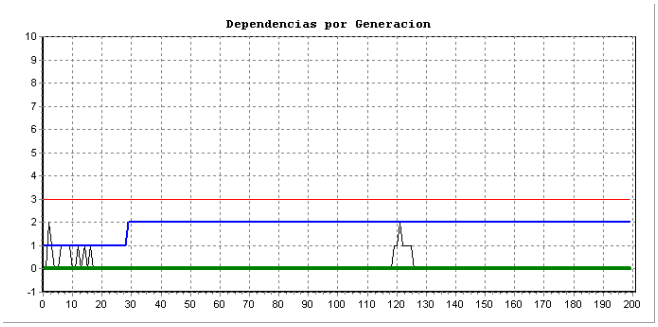



Figura 4.25: Dependencias y valores de la Celda 4

En la celda 5, figura 4.26, podemos notar un comportamiento muy interesante, ya que muestra cierta variabilidad en las dependencias de sus variables, las cuales a su vez se ven un poco reflejadas en las entradas de la compuerta, la cual, cabe resaltar, encuentra desde la primera generación. También podemos observar como el cambio de los valores de las entradas se nota a partir de la primera regeneración de la población en la generación 30.

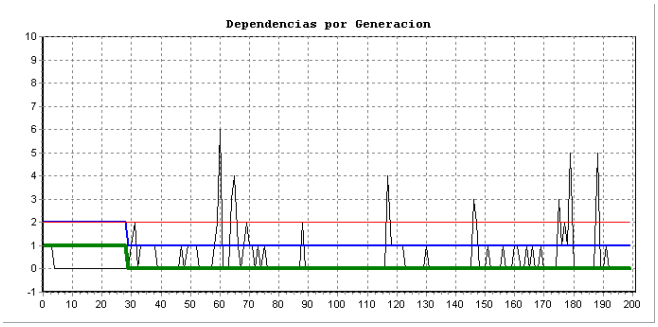



Figura 4.26: Dependencias y valores de la Celda 5

En lo referente a la celdas 6, 7 y 8, figuras 4.27, 4.28 y 4.29, podemos notar un cambio favorable en la compuerta contenida en dichas celdas, dicho cambio favorece a la solución del problema, comparando nuevamente con la gráfica que muestra el comportamiento de la función de aptitud, podremos notar cómo estos cambios en la compuerta se reflejan en un incremento del valor de aptitud.

En la figura 4.30 se muestra los valores de la celda 9. Esta es la cel-

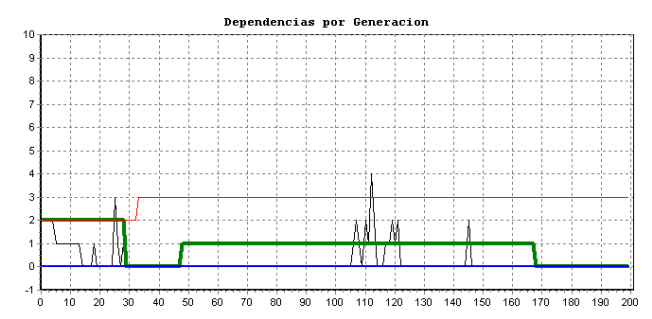



Figura 4.27: Dependencias y valores de la Celda 6

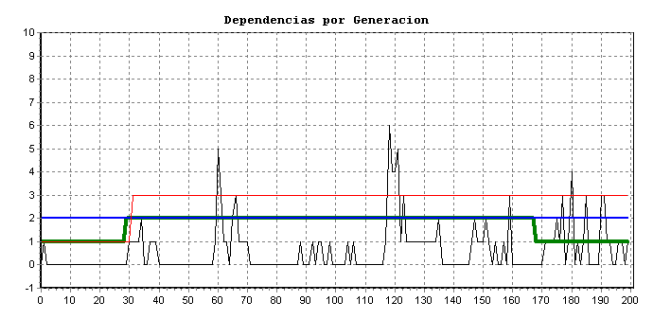



Figura 4.28: Dependencias y valores de la Celda 7

da que nos indica el renglón de salida de la matriz, la cual presenta un comportamiento muy interesante dado que desde la primera generación encuentra una compuerta que favorece la solución del problema, así como las entradas de dicha compuerta, y no la pierde en ningún momento, lo cual es interesante dado que llega a tener mucha variabilidad en cuanto a las dependencias entre las variables que conforman dicha matriz.

En las gráficas referentes a los valores que van tomando las celdas junto con el número de dependencias de éstas podemos notar cómo en casi todas las celdas las dependencias entre las variables desaparecen en las últimas generaciones. También podemos observar un comportamiento poco uniforme en dichas dependencias, esto tal vez debido a la mutación, la cual se le aplica a la población después de que ésta se ha regenerado con las dependencias de la población de la generación anterior.

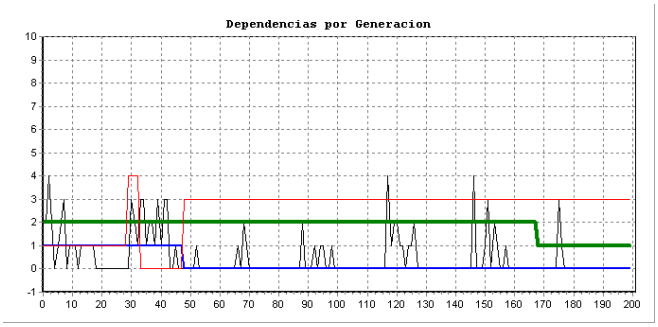



Figura 4.29: Dependencias y valores de la Celda 8

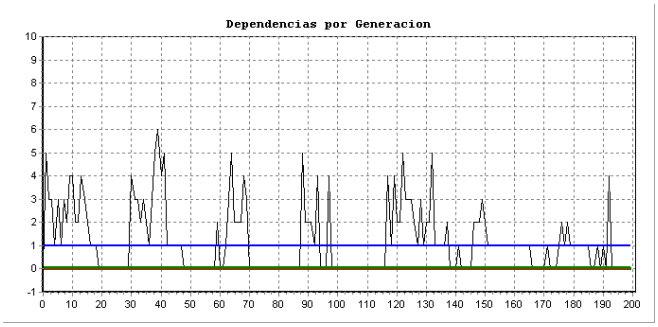



Figura 4.30: Dependencias y valores de la Celda 9

También podemos notar, en cuanto a la compuerta y las entradas contenidas en cada celda, existe una variabilidad que en algunos casos corresponde a la regeneración de la población que se hace cada 30 generaciones, pero lo que si podemos constatar es que el cambio a WIRES en algunas de las celdas que iniciaron con alguna otra compuerta se ve reflejado en la función de aptitud y las generaciones en que aparecen corresponden con los saltos que aparecen en la gráfica que muestra el comportamiento de la función de aptitud. También es importante notar, que en la celda 4 siempre hubo un WIRE, y que en la generación 30 aparecen dos WIRES, lo que provoca el cambio de la función de aptitud de 7 a 11.

A continuación se muestra la variación de la probabilidad marginal de cada una de las variables de cada celda. Primero se observarán las probabilidades de todas las variables de cada celda y posteriormente cada una por separado.

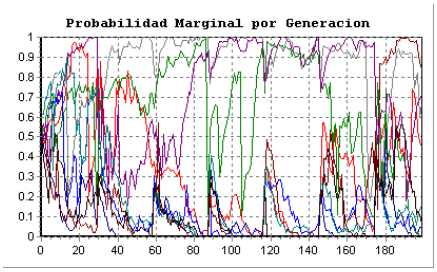



Figura 4.31: Probabilidad Marginal de las Variables de la Celda 1

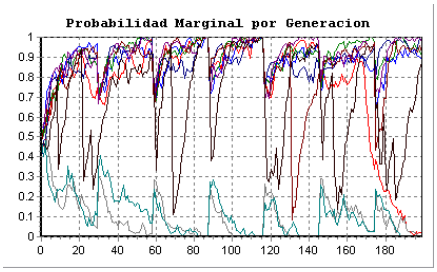



Figura 4.32: Probabilidad Marginal de las Variables de la Celda 2

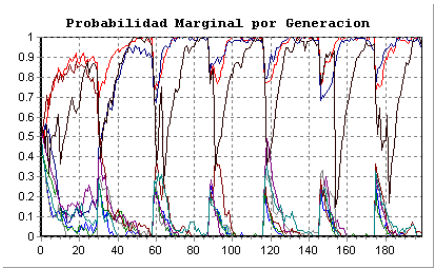



Figura 4.33: Probabilidad Marginal de las Variables de la Celda 3

En las figuras 4.40 y 4.41 se muestran las gráficas correspondientes a cada una de las variables de las celdas 1 y 4, respectivamente, ya que en las gráficas de las dependencias entre las variables de las celdas, la celda 1 y 7 tienen un comportamiento similar ya que al final de la corrida siguen conservando algunas dependencias entre sus variables, en cambio las celdas 2, 3, 4, 5, 6, 8 y 9 al final de la corrida no muestran dependencia entre sus variables.

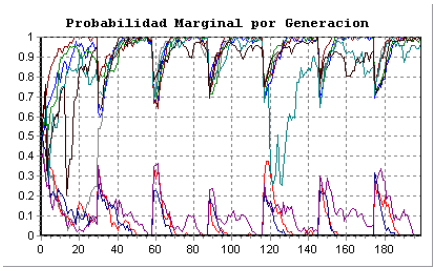



Figura 4.34: Probabilidad Marginal de las Variables de la Celda 4

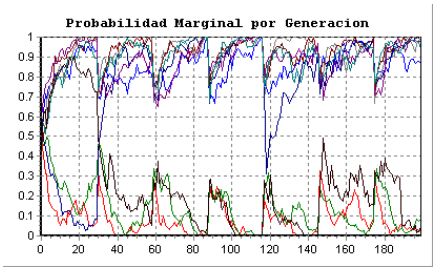



Figura 4.35: Probabilidad Marginal de las Variables de la Celda 5

Como podemos observar en las gráficas de la figura 4.41, la mayoría de las variables de la celda 4 tienen probabilidades muy extremas en cuanto a tomar un valor de uno se refiere, es decir, que la probabilidad de que los valores de la variable sean 1 es muy alta o muy baja, pero cada variable tiene una tendencia, en cambio si observamos las variables de la celda 1, figura 4.40 podemos notar que sus probabilidades de tener solo unos son muy variantes, que dichas variables no tienen una tendencia a favorecer al 1 o al 0.

Al observar el comportamiento de las probabilidades de las variables que componen cada una de las celdas de la matriz se puede decir que los saltos que se muestran el las gráficas con originados por una mutación que se implementó en el algoritmo, ya que ésta se ejecuta de manera aleatoria y sobre una población que ya fue generada por el algoritmo del poliárbol, quitando algunos de sus individuos para meter en esta nueva población mutaciones del mejor individuo de la generación anterior.

La matriz resultante queda formada como se muestra en la figura 4.42.

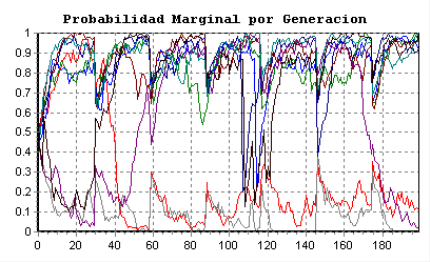



Figura 4.36: Probabilidad Marginal de las Variables de la Celda 6

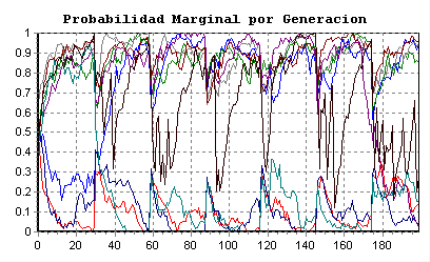



Figura 4.37: Probabilidad Marginal de las Variables de la Celda 7

De la matriz anterior, armamos el circuito resultante que satisface la función lógica que se muestra en el cuadro 4.5. Dicho circuito se muestra en la figura 4.43.

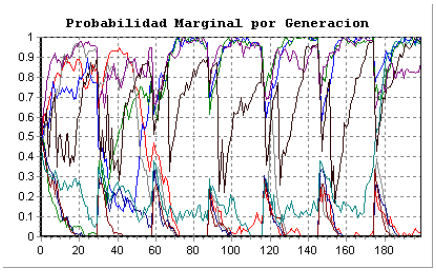



Figura 4.38: Probabilidad Marginal de las Variables de la Celda 8

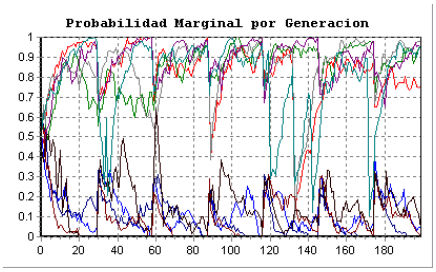



Figura 4.39: Probabilidad Marginal de las Variables de la Celda 9

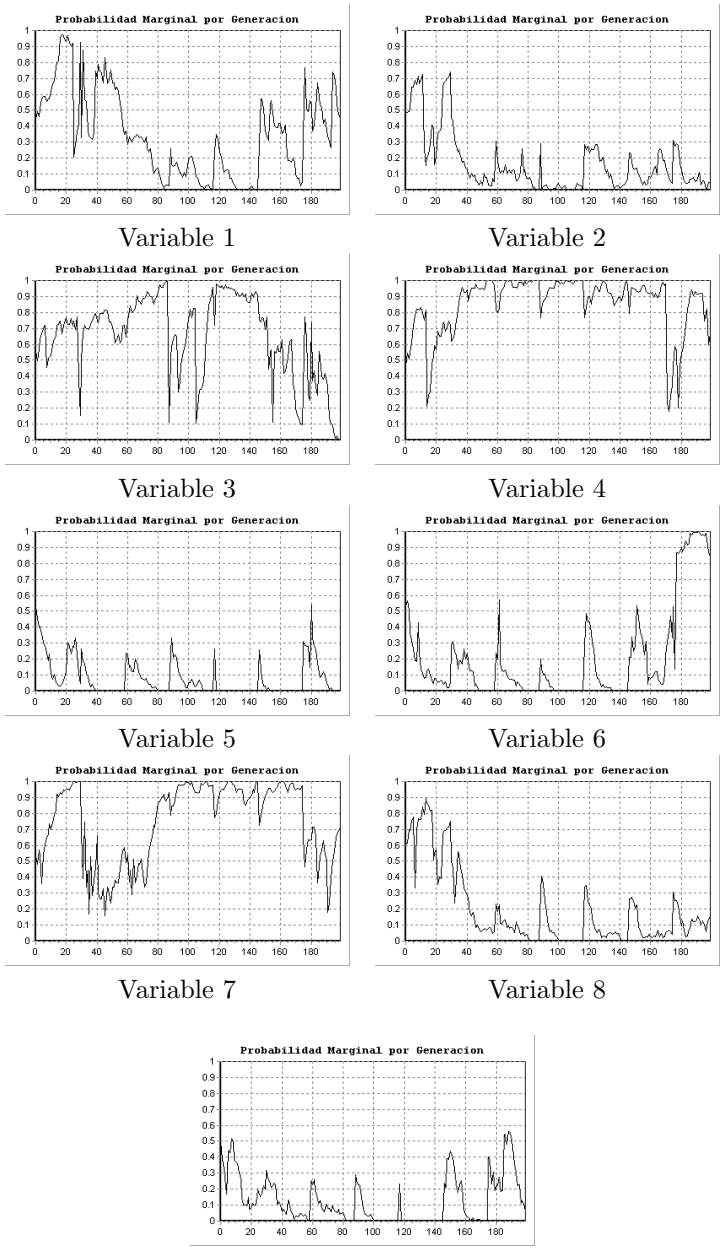



Figura 4.40: Probabilidades Marginales de las Variables de la Celda 1

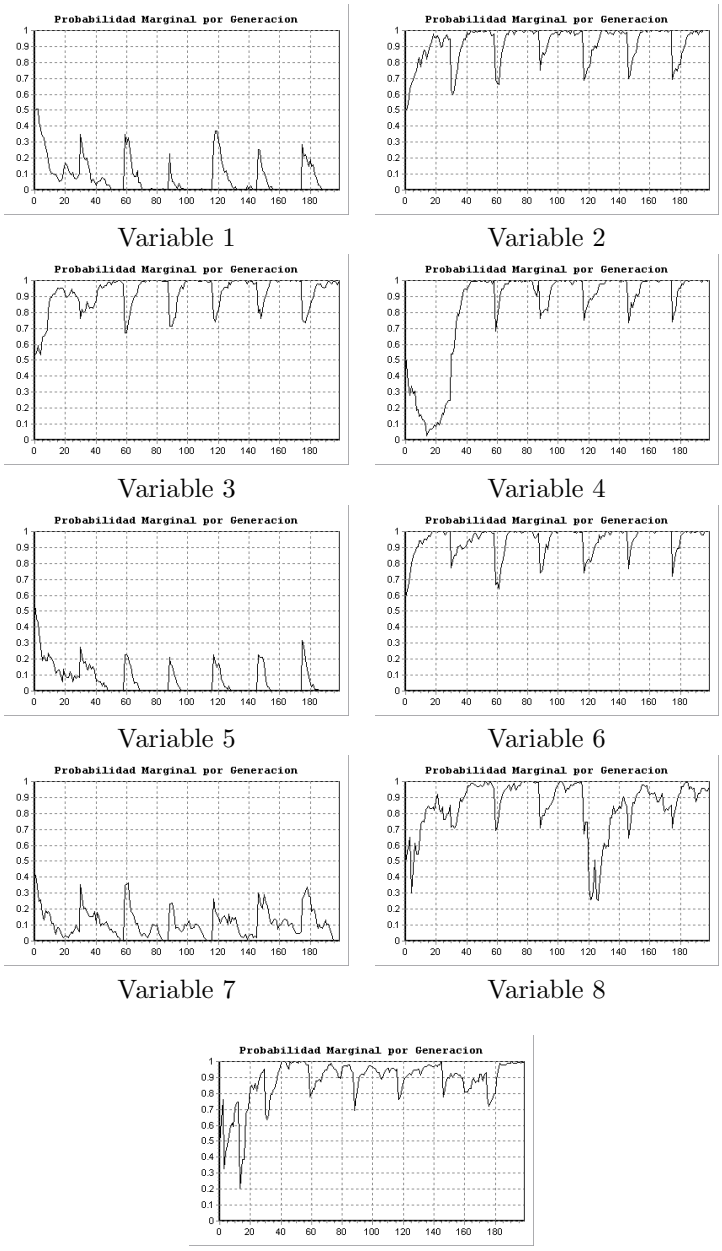



Figura 4.41: Probabilidades Marginales de las Variables de la Celda 4

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| AND | 2 | 1 | WIRE | 2 | 0 | WIRE | 2 | 1 |
| WIRE | 0 | 2 | XOR | 1 | 0 | WIRE | 0 | 1 |
| OR | 2 | 1 | WIRE | 0 | 0 | AND | 1 | 0 |

Figura 4.42: Matriz resultante del caso de prueba

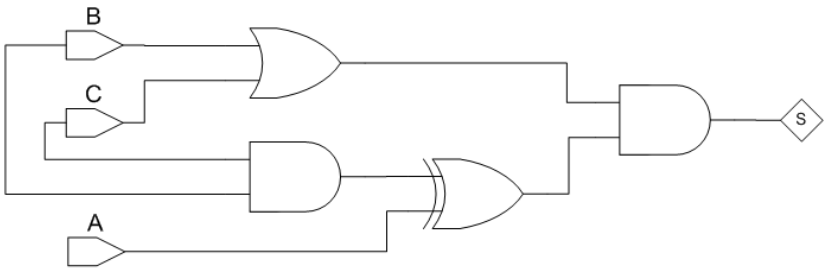



Figura 4.43: Circuito resultante para la función del cuadro 4.5

# Capítulo 5

# Experimentos y Resultados

El desempeño del algoritmo de estimación de distribución *poliárbol* para el diseño de circuitos lógicos combinatorios se evaluó mediante una serie de experimentos en donde el objetivo principal era evaluar la consistencia del algoritmo para producir circuitos factibles que minimizaran el número de compuertas utilizadas.

Cada experimento consta de una tabla de verdad de algunas variables de entrada y una o más salidas que describen el comportamiento de un circuito en su totalidad. La tabla de verdad es introducida al programa y se lleva a cabo un conjunto de corridas para analizar el comportamiento estadístico del algoritmo. Se mostraran siete circuitos con distintos grados de dificultad y por ende con una cantidad distinta de evaluaciones de la función de aptitud. Algunos de los resultados se mostrarán en tablas comparativas con otros algoritmos que hayan resuelto la función lógica de la cual se estén reportando dichos resultados.

## 5.1. Ejemplo 1

El ejemplo 1 es un circuito sencillo de cuatro entradas y una salida cuya tabla de verdad se muestra en el cuadro 5.1. Se hicieron 30 corridas con 100,000 evaluaciones de la función de aptitud en una matriz de $5 \times 5$. Los resultados comparativos con el PSO[12] obtenidos se muestran en el cuadro 5.2.

Tambien se hicieron 30 corridas con los parámetros que se muestran en el cuadro 5.3. Estos resultados no se comparan con otro algoritmo

| A | B | C | D | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Cuadro 5.1: Tabla de verdad del ejemplo 1

| **Resultados** | **PSO** | **Poliárbol** |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 6 | 6 |
| Frecuencia de la Mejor Solución | 75 % | 40 % |
| Circuitos Factibles | 100 % | 100 % |
| Promedio del Número de Compuertas | 6.75 | 6.4 |
| Promedio de la Función de Aptitud | 34.25 | 30.2 |

Cuadro 5.2: Comparación del Resultados entre el PSO y el Poliárbol para el ejemplo 1 con 100,000 evaluaciones de la función de aptitud

| Parámetros y Resultados | Poliárbol |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 30,000 |
| Tamaño de la Matriz | $4 \times 4$ |
| No. de Compuertas de la Mejor Solución | 6 |
| Frecuencia de la Mejor Solución | 36 % |
| Circuitos Factibles | 100 % |
| Promedio del Número de Compuertas | 7 |
| Promedio de la Función de Aptitud | 21 |
| Mejor Aptitud | 26 |
| Peor Aptitud | 14 |

Cuadro 5.3: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 1

debido a que no se han utilizado esos parámetros para resolver esta función lógica.

El circuito resultante del ejemplo 1 se muestra en la figura 5.1.

## 5.2. Ejemplo 2

Este segundo ejemplo tiene cuatro entradas y una salida, al igual que el ejemplo 1, y su tabla de verdad la podemos ver en el cuadro 5.4. Para resolver esta función lógica se hicieron 30 corridas, con 100,000 evaluaciones de la función de aptitud, y una matriz de $5 \times 5$. Los resultados comparativos con el PSO[12] se muestran en el cuadro 5.5.

Tambien se hicieron 30 corridas con los parámetros que se muestran en el cuadro 5.6. Estos resultados no se comparan con otro algoritmo debido a que no se han utilizado esos parámetros para resolver esta función lógica.

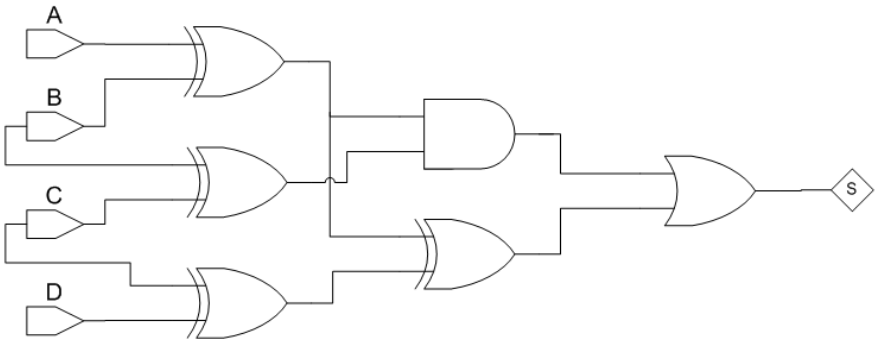



Figura 5.1: Circuito resultante para la función del ejemplo 1

| A | B | C | D | S |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 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 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Cuadro 5.4: Tabla de verdad del ejemplo 2

| Resultados | PSO | Poliárbol |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 5 | 6 |
| Frecuencia de la Mejor Solución | 20 % | 80 % |
| Circuitos Factibles | 85 % | 100 % |
| Promedio del Número de Compuertas | 10.4 | 6.8 |
| Promedio de la Función de Aptitud | 30.6 | 30.6 |

Cuadro 5.5: Comparación del Resultados entre el PSO y el Poliárbol para el ejemplo 2 con 100,000 evaluaciones de la función de aptitud

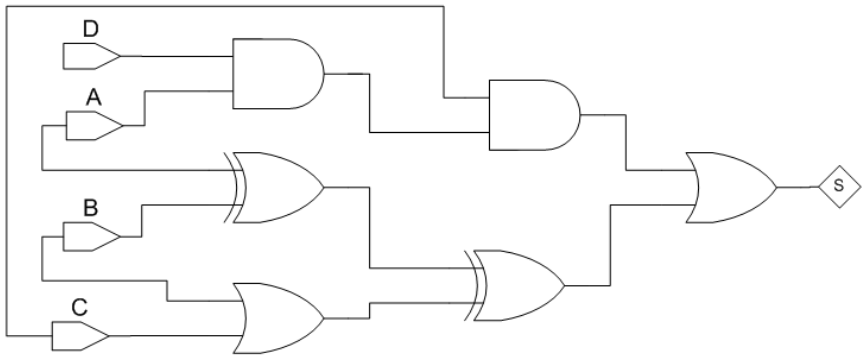



Figura 5.2: Circuito resultante para la función del ejemplo 2

El circuito resultante del ejemplo 2 se muestra en la figura 5.2.

## 5.3. Ejemplo 3

El ejemplo 3 tiene cinco entradas y una salida, y su tabla de verdad la podemos observar en el cuadro 5.7. Para resolver esta función lógica se hicieron 30 corridas, con 500,000 evaluaciones de la función de aptitud, y una matriz de $6 \times 6$. Los resultados comparativos con el PSO[12] se muestran en el cuadro 5.8.

| **Parámetros y Resultados** | **Poliárbol** |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 30,000 |
| Tamaño de la Matriz | $4 \times 4$ |
| No. de Compuertas de la Mejor Solución | 5 |
| Frecuencia de la Mejor Solución | 36 % |
| Circuitos Factibles | 50 % |
| Promedio del Número de Compuertas | 6 |
| Promedio de la Función de Aptitud | 18.8 |
| Mejor Aptitud | 25 |
| Peor Aptitud | 14 |

Cuadro 5.6: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 2

Tambien se hicieron 30 corridas con los parámetros que se muestran en el cuadro 5.9. Estos resultados no se comparan con otro algoritmo debido a que no se han utilizado esos parámetros para resolver esta función lógica.

El circuito resultante del ejemplo 3 se muestra en la figura 5.3.

## 5.4. Ejemplo 4

Este cuarto ejemplo tiene cuatro entradas y dos salidas, y su tabla de verdad la podemos ver en el cuadro 5.10. Para resolver esta función lógica se hicieron 30 corridas, con 200,000 evaluaciones de la función de aptitud, y una matriz de $5 \times 5$. Los resultados comparativos con el PSO[12] se muestran en el cuadro 5.11.

Tambien se hicieron 30 corridas con los parámetros que se muestran en el cuadro 5.12. Estos resultados no se comparan con otro algoritmo

| A | B | C | D | E | S |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 |

Cuadro 5.7: Tabla de verdad del ejemplo 3

| **Resultados** | **PSO** | **Poliárbol** |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 0 | 0 |
| Frecuencia de la Mejor Solución | 0 | 0 |
| Circuitos Factibles | 0 | 0 |
| Promedio del Número de Compuertas | 0 | 0 |
| Promedio de la Función de Aptitud | 28.95 | 29.5 |

Cuadro 5.8: Comparación del Resultados entre el PSO y el Poliárbol para el ejemplo 3 con 500,000 evaluaciones de la función de aptitud

| **Parámetros y Resultados** | **Poliárbol** |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 75,000 |
| Tamaño de la Matriz | $7 \times 7$ |
| No. de Compuertas de la Mejor Solución | 12 |
| Frecuencia de la Mejor Solución | 3 % |
| Circuitos Factibles | 28 % |
| Promedio del Número de Compuertas | 6 |
| Promedio de la Función de Aptitud | 14.33 |
| Mejor Aptitud | 69 |
| Peor Aptitud | 28 |

Cuadro 5.9: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 3

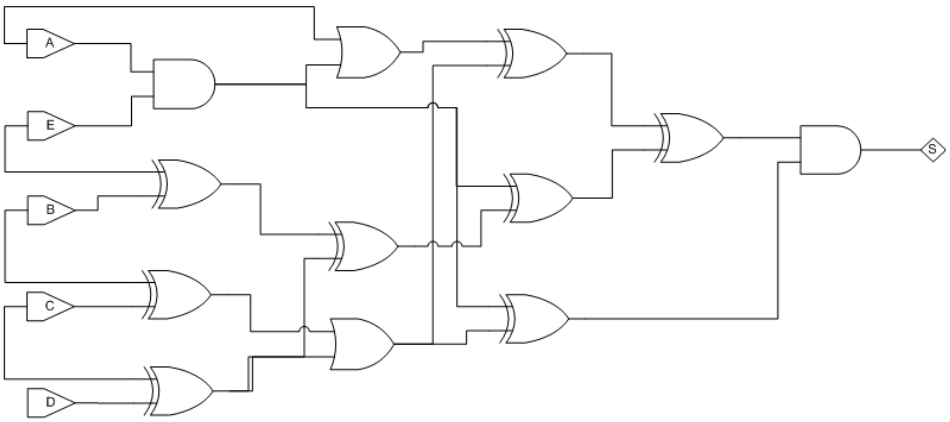



Figura 5.3: Circuito resultante para la función del ejemplo 3

| A | B | C | D | $S_0$ | $S_1$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 |

Cuadro 5.10: Tabla de verdad del ejemplo 4

| Resultados | PSO | Poliárbol |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 7 | 8 |
| Frecuencia de la Mejor Solución | 30 % | 60 % |
| Circuitos Factibles | 95 % | 60 % |
| Promedio del Número de Compuertas | 10.05 | 8 |
| Promedio de la Función de Aptitud | 46.95 | 41 |

Cuadro 5.11: Comparación del Resultados entre el PSO y el Poliárbol para el ejemplo 4 con 200,000 evaluaciones de la función de aptitud

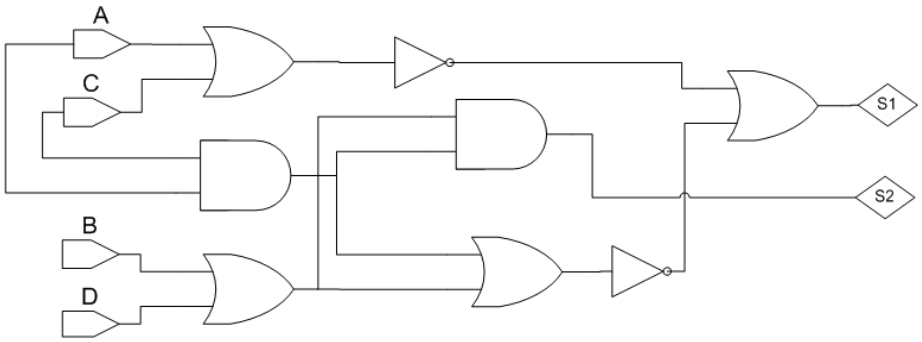



Figura 5.4: Circuito resultante para la función del ejemplo 4

debido a que no se han utilizado esos parámetros para resolver esta función lógica.

El circuito resultante del ejemplo 4se muestra en la figura 5.4.

## 5.5. Ejemplo 5

El ejemplo 5 es un circuito de cuatro entradas y tres salidas cuya tabla de verdad se muestra en el cuadro 5.13. Se hicieron 30 corridas con 500,000 evaluaciones de la función de aptitud en una matriz de $5 \times 5$. Los resultados comparativos con el PSO[12] obtenidos se muestran en el cuadro 5.14.

| Parámetros y Resultados | Poliárbol |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 90,000 |
| Tamaño de la Matriz | $7 \times 7$ |
| No. de Compuertas de la Mejor Solución | 7 |
| Frecuencia de la Mejor Solución | 3 % |
| Circuitos Factibles | 3 % |
| Promedio del Número de Compuertas | 7 |
| Promedio de la Función de Aptitud | 30.36 |
| Mejor Aptitud | 52 |
| Peor Aptitud | 29 |

Cuadro 5.12: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 4

Tambien se hicieron 30 corridas con los parámetros que se muestran en el cuadro 5.15. Estos resultados no se comparan con otro algoritmo debido a que no se han utilizado esos parámetros para resolver esta función lógica.

El circuito resultante del ejemplo 5 se muestra en la figura 5.5.

## 5.6. Ejemplo 6

El sexto ejemplo tiene dos entradas y cuatro salidas (es un decodificador 2-4) y se muestra en el cuadro 5.16.Para resolver esta función lógica se hicieron 30 corridas, con 185,400 evaluaciones de la función de aptitud, y una matriz de $5 \times 5$. Los resultados comparativos con el Ant System[4] se muestran en el cuadro 5.17.

Tambien se hicieron 30 corridas con los parámetros que se muestran

| A | B | C | D | $S_0$ | $S_1$ | $S_2$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Cuadro 5.13: Tabla de verdad del ejemplo 5

| **Resultados** | **PSO** | **Poliárbol** |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 7 | 7 |
| Frecuencia de la Mejor Solución | 10 % | 16.67 % |
| Circuitos Factibles | 55 % | 20 % |
| Promedio del Número de Compuertas | 17.15 | 7.66 |
| Promedio de la Función de Aptitud | 55.85 | 48.95 |

Cuadro 5.14: Comparación del Resultados entre el PSO y el Poliárbol para el ejemplo 5 con 500,000 evaluaciones de la función de aptitud

| **Parámetros y Resultados** | **Poliárbol** |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 62,500 |
| Tamaño de la Matriz | $4 \times 4$ |
| No. de Compuertas de la Mejor Solución | 7 |
| Frecuencia de la Mejor Solución | 6 % |
| Circuitos Factibles | 10 % |
| Promedio del Número de Compuertas | 8.33 |
| Promedio de la Función de Aptitud | 45.3 |
| Mejor Aptitud | 56 |
| Peor Aptitud | 42 |

Cuadro 5.15: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 5

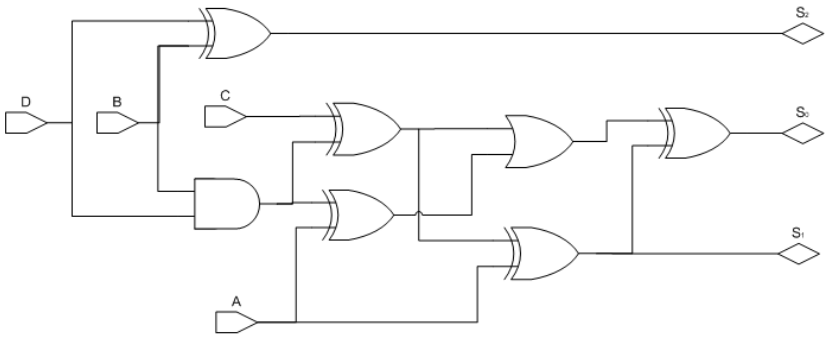



Figura 5.5: Circuito resultante para la función del ejemplo 5

| A | B | $S_0$ | $S_1$ | $S_2$ | $S_3$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Cuadro 5.16: Tabla de verdad del ejemplo 6

| Resultados | Ant System | Poliárbol |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 5 | 9 |
| Frecuencia de la Mejor Solución | 100 % | 10 % |
| Circuitos Factibles | 100 % | 10 % |
| Promedio de la Función de Aptitud | 36 | 23 |
| Mejor Aptitud | 36 | 31 |
| Peor Aptitud | 36 | 15 |

Cuadro 5.17: Comparación del Resultados entre el Ant System y el Poliárbol para el ejemplo 6 con 185,400 evaluaciones de la función de aptitud

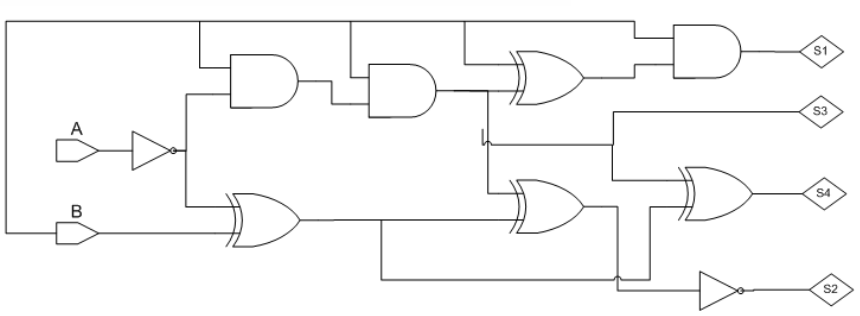



Figura 5.6: Circuito resultante para la función del ejemplo 6

en el cuadro 5.18. Estos resultados no se comparan con otro algoritmo debido a que no se han utilizado esos parámetros para resolver esta función lógica.

El circuito resultante del ejemplo 6 se muestra en la figura 5.6.

## 5.7. Ejemplo 7

El ejemplo 7 tiene cuatro entradas y una salida y se muestra en el cuadro 5.19. Para resolver esta función lógica se hicieron 30 corridas, con 185,400 evaluaciones de la función de aptitud, y una matriz de

| **Parámetros y Resultados** | **Poliárbol** |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 30,000 |
| Tamaño de la Matriz | $4 \times 4$ |
| No. de Compuertas de la Mejor Solución | 6 |
| Frecuencia de la Mejor Solución | 3 % |
| Circuitos Factibles | 16.6 % |
| Promedio del Número de Compuertas | 7.6 |
| Promedio de la Función de Aptitud | 16.26 |
| Mejor Aptitud | 26 |
| Peor Aptitud | 15 |

Cuadro 5.18: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 6

$5 \times 5$. Los resultados comparativos con el Ant System[4] se muestran en el cuadro 5.20.

Tambien se hicieron 30 corridas con los parámetros que se muestran en el cuadro 5.21. Estos resultados no se comparan con otro algoritmo debido a que no se han utilizado esos parámetros para resolver esta función lógica.

El circuito resultante del ejemplo 7 se muestra en la figura 5.7.

## 5.8. Ejemplo 8

El octavo ejemplo tiene cuatro entradas y una salida y se muestra en el cuadro 5.22.Para resolver esta función lógica se hicieron 30 corridas, con 82,400 evaluaciones de la función de aptitud, y una matriz de $5 \times 5$. Los resultados comparativos con el Ant System[4] se muestran en el cuadro 5.23.

| A | B | C | D | S |
|---|---|---|---|---|
| 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 |

Cuadro 5.19: Tabla de verdad del ejemplo 7

| **Resultados** | **Ant System** | **Poliárbol** |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 7 | 7 |
| Frecuencia de la Mejor Solución | 25 % | 60 % |
| Circuitos Factibles | 100 % | 100 % |
| Promedio de la Función de Aptitud | 33.15 | 31 |
| Mejor Aptitud | 34 | 34 |
| Peor Aptitud | 32 | 19 |

Cuadro 5.20: Comparación del Resultados entre el Ant System y el Poliárbol para el ejemplo 7 con 185,400 evaluaciones de la función de aptitud

| **Parámetros y Resultados** | **Poliárbol** |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 40,000 |
| Tamaño de la Matriz | $4 \times 4$ |
| No. de Compuertas de la Mejor Solución | 6 |
| Frecuencia de la Mejor Solución | 3 % |
| Circuitos Factibles | 26.67 % |
| Promedio del Número de Compuertas | 7.66 |
| Promedio de la Función de Aptitud | 17 |
| Mejor Aptitud | 26 |
| Peor Aptitud | 15 |

Cuadro 5.21: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 7

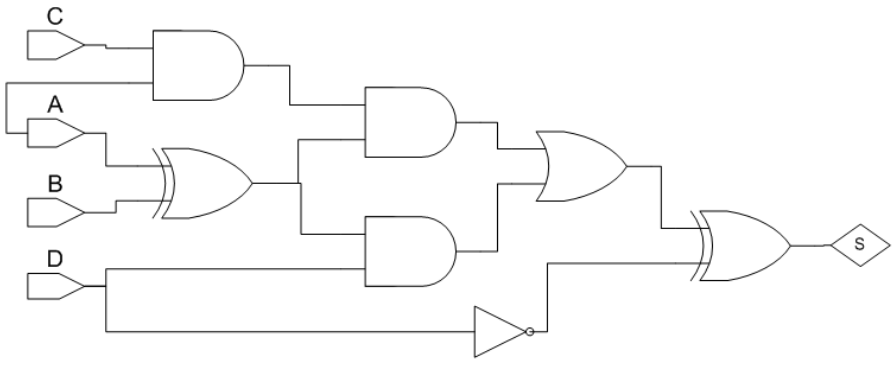



Figura 5.7: Circuito resultante para la función del ejemplo 7

| A | B | C | D | S |
|---|---|---|---|---|
| 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 |

Cuadro 5.22: Tabla de verdad del ejemplo 8

Tambien se hicieron 30 corridas con los parámetros que se muestran en el cuadro 5.24. Estos resultados no se comparan con otro algoritmo debido a que no se han utilizado esos parámetros para resolver esta función lógica.

El circuito resultante del ejemplo 8 se muestra en la figura 5.8.

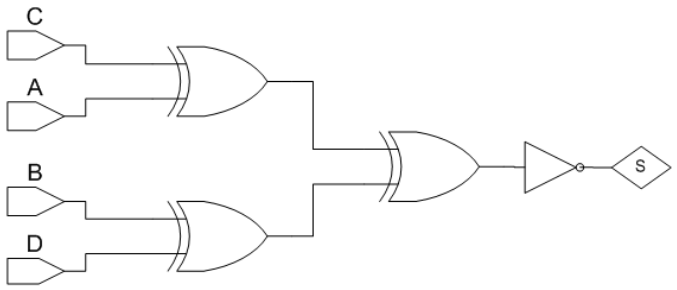



Figura 5.8: Circuito resultante para la función del ejemplo 8

| Resultados | Ant System | Poliárbol |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 4 | 4 |
| Frecuencia de la Mejor Solución | 100 % | 20 % |
| Circuitos Factibles | 100 % | 100 % |
| Promedio de la Función de Aptitud | 37 | 35 |
| Mejor Aptitud | 37 | 37 |
| Peor Aptitud | 37 | 34 |

Cuadro 5.23: Comparación del Resultados entre el Ant System y el Poliárbol para el ejemplo 8 con 82,400 evaluaciones de la función de aptitud

| Parámetros y Resultados | Poliárbol |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 40,000 |
| Tamaño de la Matriz | $4 \times 4$ |
| No. de Compuertas de la Mejor Solución | 4 |
| Frecuencia de la Mejor Solución | 56.67 % |
| Circuitos Factibles | 100 % |
| Promedio del Número de Compuertas | 5 |
| Promedio de la Función de Aptitud | 26.28 |
| Mejor Aptitud | 28 |
| Peor Aptitud | 25 |

Cuadro 5.24: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 8

| A | B | C | S |
|---|---|---|---|
| 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 |

Cuadro 5.25: Tabla de verdad del ejemplo 9

| **Resultados** | **Ant System** | **Poliárbol** |
|---|---|---|
| No. de Compuertas de la Mejor Solución | 4 | 5 |
| Frecuencia de la Mejor Solución | 85 % | 60 % |
| Circuitos Factibles | 100 % | 100 % |
| Promedio de la Función de Aptitud | 28.85 | 22.8 |
| Mejor Aptitud | 29 | 25 |
| Peor Aptitud | 28 | 18 |

Cuadro 5.26: Comparación del Resultados entre el Ant System y el Poliárbol para el ejemplo 9 con 20,600 evaluaciones de la función de aptitud

## 5.9. Ejemplo 9

El ejemplo 9 tiene tres entradas y una salida y su tabla de verdad se muestra en el cuadro 5.25.Para resolver esta función lógica se hicieron 30 corridas, con 20,600 evaluaciones de la función de aptitud, y una matriz de $5 \times 5$. Los resultados comparativos con el Ant System[4] se muestran en el cuadro 5.26.

Tambien se hicieron 30 corridas con los parámetros que se muestran en el cuadro 5.27. Estos resultados no se comparan con otro algoritmo debido a que no se han utilizado esos parámetros para resolver esta función lógica.

| **Parámetros y Resultados** | **Poliárbol** |
|---|---|
| No. de Evaluaciondes de la Función de Aptitud | 30,000 |
| Tamaño de la Matriz | $3 \times 3$ |
| No. de Compuertas de la Mejor Solución | 4 |
| Frecuencia de la Mejor Solución | 16 % |
| Circuitos Factibles | 100 % |
| Promedio del Número de Compuertas | 5 |
| Promedio de la Función de Aptitud | 11.96 |
| Mejor Aptitud | 13 |
| Peor Aptitud | 10 |

Cuadro 5.27: Parámetros y Resultados de 30 corridas independientes del Poliárbol para el ejemplo 9

El circuito resultante del ejemplo 9 se muestra en la figura 5.9.

En el cuadro 5.28 se muestra una comparación del poliárbol con el PSO y AS en cuanto al número de evaluaciones de la función de aptitud se refiere.

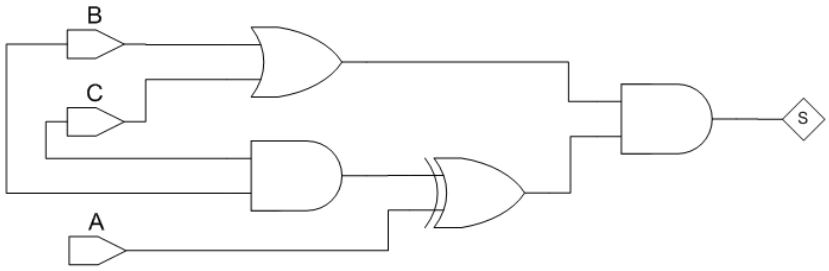



Figura 5.9: Circuito resultante para la función del ejemplo 9

| Función Lógica del Ejemplo | Poliárbol | PSO/AS |
|---|---|---|
| 1 | 30,000 | 100,000 |
| 2 | 30,000 | 100,000 |
| 3 | 75,000 | 500,000 |
| 4 | 90,000 | 200,000 |
| 5 | 62,500 | 500,000 |
| 6 | 30,000 | 185,400 |
| 7 | 40,000 | 185,400 |
| 8 | 40,000 | 82,400 |
| 9 | 30,000 | 20,600 |

Cuadro 5.28: Tabla comparativa del Poliárbol con el PSO y AS

# Capítulo 6

# Conclusiones y Trabajo Futuro

Los algoritmos de estimación de distribución (EDA) han demostrado ser herramientas poderosas para resolver problemas de optimización, ya que son capaces de obtener resultados competitivos con las técnicas evolutivas utilizadas tradicionalmente para resolver estos problemas. Lo podemos notar claramente con las funciones deceptivas, las cuales un EDA resuelve con menos dificultad que un algoritmo genético.

En este trabajo de tesis se presenta una técnica de hardware evolutivo extrínseco, que utiliza un EDA poliárbol, el cual tiene como función principal encontrar el circuito con el menor número de compuertas que cumpla con una función lógica dada.

El algoritmo propuesto trabajó con una representación binaria, con la cual no tuvo problemas para resolver aquellas funciones lógicas que contaran con una sola salida, sin importar el número de entradas, esto debido a una heurística que se implementó en el decodificador la cual busca el renglón de salida que regresa el mejor valor de la función de aptitud, y no se dejó fijo como se había estado haciendo hasta ahora.

Una de las ventajas que tiene este algoritmo es que requiere de un número menor de evaluaciones de la función de aptitud para resolver las funciones lógicas que se usaron para probar el algoritmo, además de que la mayoría de dichas funciones las resolvió con matrices de menor tamaño, tal como se puede observar en el cuadro 5.28.

Este algoritmo tuvo como principal problema el tiempo necesario de ejecución del algoritmo de aprendizaje, el cual se mencionó a detalle en el capítulo cuatro. Otro problema que se tuvo con el algoritmo se vio reflejado en las funciones lógicas que tienen más de una salida, ya que independientemente del número de evaluaciones de la función de aptitud llegó a una solución óptima en pocas ocasiones. Cabe mencionar que para estos problemas los renglones de salida se dejaron fijos, y no se aplicó la misma heurística que en el caso de los problemas con una salida.

Otro aspecto importante de este trabajo de tesis es el hecho de que por primera vez se implementa un EDA en el Hardware Evolutivo, además de que según lo observado en el capítulo cinco se obtuvieron buenos resultados, además de la inclusión de un operador de mutación en el algoritmo y la regeneración del 30 % de la población cada 30 generaciones con la finalidad de mantener la diversidad y asi poder explorar el espacio de búsqueda.

En esta propuesta se presentó el aspecto de las dependencias entre las variables que tiene cada una de las celdas de la matriz utilizada para la representación de los circuitos, y como dicha dependencia juega un papel importante en el resultado que se presenta como solución al problema planteado, ésto podría deberse a que, como se ve claramente, si una variable tiende a cambiar para mejorar el valor de la función de aptitud, las variables que dependen de ella también cambian con la misma finalidad, con lo cual podemos obtener un mejor resultado que resuelve la función lógica planteada, cosa que no sucede con el algoritmo genético, ya que como vimos en la sección 4.3 éste no utiliza las dependencias entre sus variables, lo cual nos lleva a un resultado no muy bueno del problema y a un número mayor de evaluaciones de la función de aptitud.

En cuanto al trabajo futuro que se puede hacer en torno a la propuesta actual es buscar una manera de optimizar el resultado a las funciones lógicas que presentan más de una salida. Otro aspecto importante que se puede estudiar es el comportamiento de las dependencias existentes entre las variables que forman parte de los individuos de la población procesada por el algoritmo, ya que tal vez si se oberva a detalle dicho

comportamiento se puedan llegar a resultados óptimos a los presentados en este trabajo de tesis, además de que se pueden explorar más algoritmos de estimación de ditribución ya que esta primer propuesta tuvo éxito resolviendo la mayoría de los problemas que se le plantearon.

# Bibliografía


[1] Arturo Hernández Aguirre. *Evolvable Hardware Techniques for Gate-Level Synthesis of Combinational Circuits*, chapter 12. Springer Verlag, 2004.

[2] David Beasley. An overview of genetic algorithms: Research topics. 1994.

[3] C.K. Chow and C.N. Liu. Approximating discrete probabilty distributions with dependence trees. *IEEE Transactions on Information Theory*, 1968.

[4] Carlos A. Coello Coello, Arturo Hernández Aguirre, Rosa Laura Zavala Gutiérrez, and Benito Mendoza García. Automated design of combinational logic circuits using the ant system. *ENGINEERING OPTIMIZATION*, 2001.

[5] Jeremy S. de Bonet, Charles L. Isbell, and Paul Viola. Mimic: Finding optima bye estimating probability densities. *Advances in Neural Information Processing Systems*, 1997.

[6] Benito Mendoza García. Uso del sistema de la colonia de hormigas para optimizar circuitos lógicos combinatorios. Tesis de maestria, CINVESTAV-IPN, Mayo 2001.

[7] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT PRESS, 1992.

[8] Cruz Pérez Javier. Plataforma en java para el diseño de circuitos lógicos combinatorios, experimentando con diferentes operadores genéticos. Tesis de licenciatura, Universidad Tecnológica de la Mixteca, Julio 2003.

[9] Finn V. Jensen. *Bayesian Networks and Decision Graphs*. Springer, 2001.

[10] Pedro Larrañaga. *Estimation of distribution algorithms : a new tool for evolution computation*. Kluwer Academic Pub., 2002.

[11] Erika Hernández Luna. Diseño de circuitos lógicos combinatorios usando optimización mediante cúmulos de partículas. Tesis de maestria, CINVESTAV-IPN, Febrero 2004.

[12] Erika Hernández Luna, Carlos A. Coello Coello, and Arturo Hernández Aguirre. A comparative study of encodings to design combinational logic circuits using particle swarm optimization. 2004.

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

[14] Eduardo Morales and Jesús González. Aprendizaje computacional. Se puede encontrar en http://ccc.inaoep.mx/ emorales/Cursos/NvoAprend/, 2007.

[15] Marta Rosa Soto Ortiz. *Un estudio sobre los Algoritmos Evolutivos con Estimación de Distribuciones badados en Poliárboles y su costo de evaluación*. Tesis de doctorado, Instituto de Cibernética, Matemática y Física, Julio 2003.

[16] Martin Pelikan and Heinz Mühlenbein. The bivariate marginal distribution algorithm. 1999.

[17] Mauricio Javier Soullier. Algoritmos genéticos y sistemas evolucionarios en configuraciones determinísticas. 2001.

# Apéndice A

# SRS

# Documento de Especificación de Requerimientos de Software

## A.1. Introducción

### A.1.1. Propósito

El diseño de circuitos lógicos combinatorios es un problema particular del diseño de circuitos electrónicos que tiene una complejidad tal que ha permanecido como un problema de investigación abierto a lo largo del tiempo.

El objetivo en el proceso de diseño de circuitos lógicos combinatorios es encontrar la expresión que proporcione el comportamiento deseado y que además minimice cierto criterio. En el caso de este trabajo el criterio a minimizarse es el costo del circuito, medido en términos del número necesario de compuertas para un diseño dado.

La técnica utilizada para la minimización del criterio planteado se conoce como Poliárbol, que es un Algoritmo de Estimación de Distribución (EDA).

### A.1.2. A quien va dirigido y recomendaciones de lectura

Este documento va dirigido al usuario final de la aplicación con la finalidad de que entienda la funcionalidad de la aplicación y pueda darle un uso adecuado para conseguir el objetivo buscado inicialmente. También va dirigido a aquellas personas que quieran modificar o implementar nuevas funciones a esta aplicación con la finalidad de hacerla más robusta, este documento les permitirá saber como están implementadas la mayoría de las funciones para que puedan ser reutilizables.

### A.1.3. Alcance del Proyecto

El presente proyecto es requisito para conseguir el título de Maestría en Ciencias de la Computación, el cual tiene una duración de 12 meses, a partir de Agosto de 2006 hasta Agosto de 2007, en donde la aplicación final buscará satisfacer lo siguiente:

1. Hacer una síntesis correcta de las funciones lógicas que se le introduzcan utilizando el menor número de compuertas lógicas.
2. Tener un código con modularidad, comentado y fácil de entender para futuros programadores.
3. Código reutilizable.
4. Tener una interfaz amigable para el usuario final.

### A.1.4. Referencias

Las referencias se pueden encontrar en la Bibliografía.

## A.2. Descripcion del Proyecto

### A.2.1. Perspectiva del Producto

La aplicación tiene como objetivos principales primeramente el realizar una minimización factible de nuestro criterio a optimizar, que en este caso es el número de compuertas necesarias para satisfacer una función lógica dada la cual esta representada por una tabla de verdad.

El segundo objetivo es que la aplicación sea reutilizable en cuanto a la implementación de otros algoritmos con los cuales se quieran hacer pruebas o tratar de optimizar el criterio dado anteriormente en la definición del problema.

### A.2.2. Aspectos del Producto

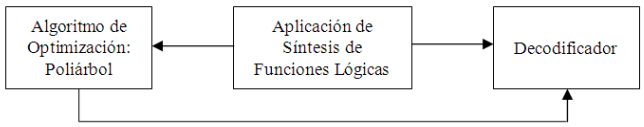



Figura A.1: Diagrama de funcionamiento del Software

Se cuenta con una aplicación que tiene dos librerías principales, la primera es la de Poliárbol, la cual contiene todas las funciones necesarias para la optimización del criterio a minimizar, y la segunda es la librería Decodificador, la cual evalúa una cadena binaria y nos devuelve un valor que corresponde al valor de la función de aptitud en dicha cadena que equivale a un individuo de la población. Una representación gráfica de el proceso mencionado se muestra en la figura A.1.

### A.2.3. Clases de Usuarios y Características

El usuario de la aplicación es una persona encargada de evaluar que el funcionamiento de la misma sea el correcto. Otro tipo de usuarios podrían ser aquellos que quieran obtener las compuertas necesarias para satisfacer una determinada tabla de verdad, que representa la función lógica que se desea optimizar, así como su interconexión.

### A.2.4. Ambiente de Operación

Este sintetizador de funciones lógicas operará en una computadora que será operada por un usuario que desee hacer uso de él.

#### Recursos de Software

Sólo se requiere que la máquina tenga instalado el compilador C++ Builder 5.1.

**Recursos de Hardware**

Se requiere de una máquina que tenga como mínimo 256 MB en RAM y un disco duro mayor de 20 GB.

### A.2.5. Restricciones de Diseño e Implementación

Esta aplicación requiere de mucha memoria y tiempo del procesador debido a la cantidad de cálculos que tiene que hacer para llegar a una solución óptima, por lo que no se deben tener en ejecución otros programas al mismo tiempo que se corre esta aplicación, esto debido a que abarca mucha de la utilización del CPU y la memoria RAM requerida es muy grande.

### A.2.6. Documentación del Usuario

Se le explicará al usuario el funcionamiento del software, así como los parámetros necesarios para poder hacerlo funcionar. Además se dejarán varios documentos que explicarán a detalle el funcionamiento de este software, así como todas las consideraciones que se tomaron en cuenta para su desarrollo, todo ésto se encuentra en los primeros cuatro capítulos de la tesis.

Se dejarán también algunos ejemplos con los cuales se corrió dicho software con la finalidad de que se verifique el correcto funcionamiento de este programa y dar una idea de los parámetros que se requieren para echarlo a andar.

### A.2.7. Suposiciones y Dependencias

Este software tiene una fuerte dependencia al compilador C++ Builder 5.1, el cual deberá estar instalado en la máquina en la cual se vaya a ejecutar esta aplicación. Otro aspecto importante de esta aplicación es que es muy sensible a los parámetros que son introducidos por el usuario, ante una mala definición de parámetros los resultados pueden no ser los esperados.

## A.3. Aspectos del Sistema

### A.3.1. Modularidad

Después de exhaustivas pruebas con dos algoritmos diferentes en las cuales no se vieron buenos resultados para la resolución de nuestro problema planteado inicialmente, se decidió hacer este documento en base solo al algoritmo del Poliárbol, dejando abierta la posibilidad de implementar otros algoritmos para verificar el correcto funcionamiento de la aplicación. En este caso, los algoritmos elegidos no fueron los mejores y dado el tiempo tan limitado que se tiene para estas pruebas, no se pudieron implementar otros algoritmos. Para la aplicación final el caso de uso correspondiente se muestra el figura A.2.

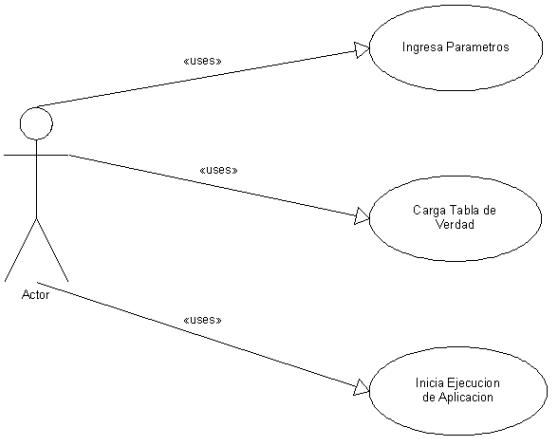



Figura A.2: Caso de Uso del Software

A continuación se ven a detalle los módulos principales de esta aplicación los cuales son usados en los tres casos presentados en el caso de uso anterior.

### A.3.2. Módulo 1: Decodificador

#### Descripción y Prioridad

La finalidad de este módulo es la de regresar un circuito al usuario indicando el tipo de compuertas que requerirá dicho circuito así como

sus entradas, además de regresar el resultado de evaluar la función lógica dada por el usuario en dicho circuito.

#### Acciones del Usuario Respuestas del Sistema

Lo único que debe hacer el usuario es cargar una tabla de verdad y especificar algunos parámetros que son requeridos por este módulo. El sistema regresará un circuito evaluado en la función lógica.

#### Requerimientos Funcionales

1. Regresar un circuito en forma de matriz indicando que compuertas lógicas se usarán además de sus entradas.
2. Regresar el resultado de evaluar dicho circuito en la función lógica dada por el usuario.

#### Variables y Funciones que componen al Módulo Decodificador

A continuación se da una breve descripción de los metodos y atributos de la clase que compone el módulo decodificador.

### Variables

- **no_ent.** Variable de tipo entero que indica el número de entradas que contiene la Tabla de Verdad.
- **ren_tv.** Variable de tipo entero que indica el número de renglones que contiene la Tabla de Verdad.
- **sal_tv.** Variable de tipo entero que indica el número de salidas que contiene la Tabla de Verdad.
- **no_comp.** Variable de tipo entero que indica el número de compuertas que se utilizarán para resolver la función lógica.
- **no_col.** Variable de tipo entero que indica el número de columnas de la matriz en donde se generará el circuito.
- **no_ren.** Variable de tipo entero que indica el número de renglones de la matriz en donde se generará el circuito.
- **a_ind.** Variable de tipo entero que define el tamaño del individuo con el que se trabajará en el módulo Poliárbol.

- **tabla.** Matriz de tipo entero en donde se almacena la Tabla de Verdad que es la función lógica que se busca resolver.
- **comp_ent.** Matriz de tipo entero que guarda el circuito decodificado de una cadena de ceros y unos a una matriz que contiene números enteros que representan compuertas y entradas, las cuales son necesarias para evaluar la función lógica.
- **grafica.** Matriz de tipo entero que guarda los valores de compuertas que va tomando comp_ent.
- **peso.** Matriz de tipo flotante que guarda el valor ponderado calculado tomando en cuenta el número de ceros o unos que haya en las salidas de la tabla de verdad.
- **r_sal.** Arreglo de tipo entero que guarda los posibles renglones de salida de la matriz que contiene al circuito en el que evaluamos la función lógica.
- **comp_char.** Matriz de tipo char que guarda los nombres de las compuertas que componen al circuito que regresa el decodificador.
- **and.** Variable de tipo booleano que indica si se va a utilizar la compuerta and o no.
- **or.** Variable de tipo booleano que indica si se va a utilizar la compuerta or o no.
- **xor.** Variable de tipo booleano que indica si se va a utilizar la compuerta xor o no.
- **wire.** Variable de tipo booleano que indica si se va a utilizar la compuerta wire o no.
- **not.** Variable de tipo booleano que indica si se va a utilizar la compuerta not o no.

**Funciones**

- **obt_tabla_verdad.** Función de tipo void que permite cargar la tabla de verdad correspondiente a la función lógica que se desea optimizar.
- **decodifica.** Función de tipo void que convierte una cadena binaria en un circuito lógico combinatorio utilizando la configuración fijada previamente por el usuario.

- **evaluar.** Función de tipo void que evalúa el circuito decodificado en la tabla de verdad previamente cargada por el usuario y regresa el número de coincidencias del circuito con los renglones de la tabla de verdad.
- **evalua_and.** Función de tipo entero que evalúa la función And entre dos valores los cuales pueden tomar valor de 0 ó 1.
- **evalua_or.** Función de tipo entero que evalúa la función Or entre dos valores los cuales pueden tomar valor de 0 ó 1.
- **evalua_xor.** Función de tipo entero que evalúa la función Xor entre dos valores los cuales pueden tomar valor de 0 ó 1.
- **evalua_not.** Función de tipo entero que evalúa la función Not entre dos valores los cuales pueden tomar valor de 0 ó 1.
- **evalua_wire.** Función de tipo entero que evalúa la función Wire entre dos valores los cuales pueden tomar valor de 0 ó 1.
- **convierte.** Función de tipo entero que convierte un valor binario a su valor entero.
- **cuenta_wires.** Función de tipo entero que cuenta el número de compuertas wires que hay en el circuito resultante que satisface la tabla de verdad.
- **buscar_menor.** Función de tipo entero que busca el elemento mayor de un arreglo, regresando la posición de dicho elemento.
- **ordena_indices.** Función de tipo void que ordena los índices de un arreglo el cual también es ordenado de mayor a menor.
- **busca_indice.** Funcion de tipo booleana que busca un número dentro de un arreglo, regresando TRUE si lo encuentra y FALSE en caso contrario.
- **inic_var.** Función de tipo void que inicializa las variables necesarias para el funcionamiento de este módulo. Esta es una de las funciones principales de la aplicación ya que recibe los parámetros fijados por el usuario y permite así llevar a cabo la solución del problema. Los parámetros que debe recibir y el orden en el que se deben pasar se muestra a continuación: *inic_var* **(# columnas, # renglones, # compuertas, bool and, bool or, bool xor, bool wire, bool not)**.

- **pedir_memoria.** Función de tipo void que pide la memoria necesaria para las variables utilizadas en este módulo.
- **regr_memoria.** Función de tipo void que regresa la memoria correspondiente a las variables de este módulo pedida al principio de la ejecución de la aplicación.

### A.3.3. Módulo 2: Poliárbol

#### Descripción y Prioridad

La finalidad de este módulo es la de darle al módulo del decodificador una cadena de ceros y unos que dicho módulo transforme en un circuito, esta cadena es generada mediante un algoritmo de estimación de distribución.

#### Acciones del Usuario / Respuestas del Sistema

El usuario dará a este módulo los parámetros necesarios para su funcionamiento.

#### Requerimientos Funcionales

1. Generar una cadena binaria de ceros y unos que al ser decodificada regrese un circuito que sea una solución óptima a la función lógica dada por el usuario.

#### Variables y Funciones que componen al Módulo Poliárbol

A continuación se da una breve descripción de los metodos y atributos de la clase que compone el módulo poliárbol.

#### Variables

- **ind.** Matriz de tipo flotante que representa nuestra población,, en donde cada renglón es un individuo.
- **muestra.** Matriz de tipo flotante que es un porcentaje de individuos tomados de la población en donde cada columna representa una variable aleatoria, que tiene tantos valores como individuos tomados para la muestra.

- **hijos.** Matriz de tipo flotante del mismo tamaño que la matriz ind la cual guarda temporalmente a la nueva población y a la cual se le aplica una pequeña mutación a algunos de sus individuos.
- **prob_conj11.** Matriz de tipo flotante que guarda las probabilidades conjuntas de dos variables de la muestra cuando ambas tienen valor 1.
- **prob_conj10.** Matriz de tipo flotante que guarda las probabilidades conjuntas de dos variables de la muestra cuando una tiene valor 1 y la otra valor 0.
- **prob_conj01.** Matriz de tipo flotante que guarda las probabilidades conjuntas de dos variables de la muestra cuando una tiene valor 0 y la otra valor 1.
- **prob_conj00.** Matriz de tipo flotante que guarda las probabilidades conjuntas de dos variables de la muestra cuando ambas tienen valor 0.
- **prob_cond.** Matriz de tipo flotante que guarda la probabilidad condicional existente entre dos variables.
- **info_mutua.** Matriz de tipo flotante que guarda la información mutua existente entre dos variables.
- **lista.** Matriz de tipo flotante que guarda pares de variables entre las cuales existe una dependencia.
- **caminos.** Matriz de tipo flotante en la cual se calcula una matriz de caminos correspondiente a un algoritmo de estructuras de datos, en este caso, de grafos.
- **depend.** Matriz de tipo flotante que guarda el valor de dependencia que existe entre varias variables tomadas en pares.
- **g_dir.** Matriz de tipo flotante que guarda las direcciones de las aristas del grafo que se genera con el algoritmo, que equivalen a las dependencias que existen entre las variables de la muestra.
- **grafo.** Matriz de tipo flotante que corresponde a la matriz de adyacencia que se genera por la teoría de grafos.
- **prob_marg.** Arreglo de tipo flotante que almacena las probabilidades marginales de las columnas de la muestra.

- **f_apt.** Arreglo de tipo flotante que guarda el valor de los individuos de la matriz ind evaluados en la función lógica.
- **fa_hijos.** Arreglo de tipo flotante que guarda el valor de los individuos de la matriz hijos evaluados en la función lógica.
- **porcen.** Valor de tipo flotante que indica el porcentaje de los individuos que se tomarán para la muestra.
- **mejores.** Matriz de tipo flotante que guarda a los mejores individuos de cada generación.
- **mejor.** Arreglo de tipo flotante que guarda los valores de los mejores individuos evaluados en la función lógica.
- **prob_conj3.** Arreglo de tipo flotante que guarda la probabilidad conjunta de tres variables.
- **info_mutua3.** Arreglo de tipo flotante que guarda la información mutua existente entre tres variables.
- **dep_g.** Arreglo de tipo flotante que almacena las dependencia global que existe entre los elementos contenidos en la matriz lista.
- **regenera.** Matriz de tipo flotante que almacena temporalmente a la población generada con la distribución calculada por el algoritmo poliárbol, para copiarse posteriormente a la matriz hijos.
- **no_bit.** Valor de tipo entero que me indica el número de bits por variable que sirve para definir el número de columnas de cada individuo.
- **no_var.** Valor de tipo entero que indica el número de variables que también sirve para definir el número de columnas de cada individuo.
- **no_ind.** Valor de tipo entero que me dice el número de individuos de la población.
- **no_gen.** Valor de tipo entero que me indica el número de generaciones, es decir el número de veces que se ejecutará el algoritmo.
- **t_mues.** Valor de tipo entero que me indica el número de individuos que tomaré para la muestra.
- **gen.** Valor de tipo entero que me indica la generación actual.

- **tam_lista.** Valor de tipo entero que me indica el tamaño de la lista que contiene los pares de variables que quedarán enlazados.
- **conecta.** Variable de tipo decodificador que me permite crear un vínculo entre los dos módulos principales de esta aplicación.

**Funciones**

- **genera_poblacion.** Función de tipo void que genera aleatoriamente a la población inicial además de evaluarla en la función lógica que se busca resolver.
- **prob_marginal.** Función de tipo void que calcula la probabilidad marginal de cada una de las columnas de una muestra tomada de la población actual.
- **prob_conjunta.** Función de tipo void que calcula las probabilidades conjuntas entre dos columnas de todas las existentes en la muestra y las va guardando en una matriz.
- **sel_muestra.** Función de tipo void que selecciona un porcentaje de individuos de la población actual a los cuales se les estimará una distribución y con los cuales se regenerará la nueva población. A este porcentaje se le conoce como muestra.
- **inf_mutua.** Función de tipo void que calcula la información mutua existente entre todas las variables de la muestra tomadas de dos en dos, guardando este valor en una matriz.
- **ordena.** Función de tipo void que ordena un arreglo de mayor a menor.
- **algoritmo.** Función de tipo void que manda llamar a todas las funciones necesarias para ejecutar el algoritmo conocido como EDA (Estimation Distribution Algorithm). Esta es una de las funciones principales de este módulo.
- **copia_poblacion.** Función de tipo void que copia la población generada por el poliárbol en la población anterior, con la finalidad de repetir todo el procedimiento de nuevo pero con la nueva población.
- **busca_indice.** Función de tipo bool que busca un número en un arreglo, regresando un valor trae si dicho número se encuentra en ese arreglo.

- **checa_dep.** Función de tipo void que busca valores mayores o iguales a un umbral en la matriz generada por la función inf_mutua. Si son mayores o iguales entonces pone en una matriz un 1 en donde se cumpla la condición anterior. Dicha matriz se llena inicialmente con ceros.

- **prob_conj3.** Función de tipo void que calcula la probabilidad conjunta entre tres variables, las cuales se le indican a esta función.

- **inf_mutua3.** Función de tipo void que calcula la información mutua existente entre tres columnas de la muestra, las cuales se le indican a esta función.

- **recortar_lista.** Función de tipo void que quita elementos de una matriz si se cumple una condición dada.

- **inserta_aristas.** Función de tipo void que llena una matriz conocida como de adyacencia, esto usando la teoría de grafos de que no debe contener ciclos.

- **matriz_caminos.** Función de tipo void que genera una matriz de caminos con la finalidad de evitar ciclos en un grafo.

- **cabeza_cabeza.** Función de tipo void que busca los nodos cabeza - cabeza que son los elementos más importantes de este algoritmo. Si encuentra un nodo cabeza - cabeza modifica la matriz de adyacencia.

- **alg_poliarbol.** Función de tipo void que manda llamar a las funciones necesarias para ejecutar el algoritmo conocido como Poliárbol que forma parte del EDA que contiene este módulo.

- **copia_muestra.** Función de tipo void que copia la muestra seleccionada a otro espacio de memoria.

- **regenera_pob.** Función de tipo void que genera una nueva población en la base a la distribución calculada con el algoritmo Poliárbol.

- **copia_hijos.** Función de tipo void que copia la población generada en la matriz hijos, la cual es la nueva población.

- **evalua_hijos.** Función de tipo void que evalúa la población hijos en la función lógica para calcular su función de aptitud.

- **elitismo.** Función de tipo void que selecciona a los mejores individuos de la población actual y de la anterior y los pone en la nueva población.
- **buscar_mayor.** Función de tipo entero que busca el elemento mayor de un arreglo regresando la posición en la que se encuentra.
- **buscar_menor.** Función de tipo entero que busca el elemento menor de un arreglo regresando la posición en la que se encuentra.
- **mutar_hijos.** Función de tipo void que hace mutaciones sobre el mejor individuo de la población hijos. Esto con la finalidad de aumentar el espacio de búsqueda del óptimo.
- **inicializacion.** Función de tipo void que es otra de las funciones más importantes de la aplicación ya que esta función recibe los valores iniciales y necesarios para el correcto funcionamiento de este módulo, y en conjunto con la función inic_var del módulo Decodificador dan el correcto funcionamiento de esta aplicación. Los parámetros que recibe se muestran a continuación, así como el orden en el que deben ser pasados: *inicializacion*(**# variables, #bits por variable, # individuos, # generaciones, % muestra**).
- **pedir_memoria.** Función de tipo void que pide la memoria necesaria para las variables utilizadas en este módulo.
- **devolver_memoria.** Función de tipo void que regresa la memoria correspondiente a las variables de este módulo pedida al principio de la ejecución de la aplicación.

## A.4. Requerimientos de Interfases Externas

### A.4.1. Interfases de Usuario

La interfaz final de la aplicación se muestra en la figura A.3. El usuario deberá llenar algunos parámetros los cuales básicamente son: número de columnas, número de renglones, número de individuos, número de generaciones, porcentaje de la muestra y número de corridas, además de seleccionar el número de compuertas que desea usar para hacer una búsqueda de la solución, así como cargar la tabla de verdad correspondiente a la función lógica que se quiere minimizar.

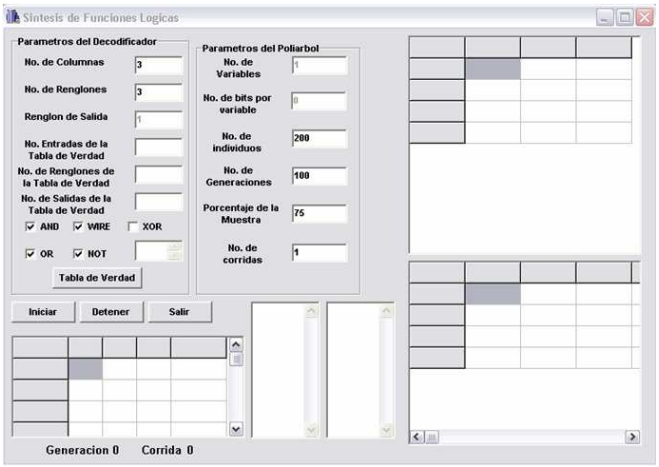



Figura A.3: Interfaz del Programa

### A.4.2. Interfases de Software

Se pretende hacer una conexión de esta aplicación con algún software orientado al diseño de circuitos lógicos como Workbench o VeriLog. Se hicieron algunas pruebas con Workbench, pero por falta de tiempo no se logró mucho, solo podemos verificar que los circuitos arrojados por este optimizador de funciones lógicas funcionan correctamente de acuerdo a lo establecido por la tabla de verdad.

## A.5. Otros Requerimientos No Funcionales

### A.5.1. Requerimientos de Reutilización de Código

Este código es reutilizable, cuenta con tres funciones principales, las cuales se explicaron a detalle en la sección 3 de este documento en las cuales se piden algunos parámetros que tienen que ser dados por el usuario para poder funcionar.

Esto se hace con la finalidad de que se puedan implementar con facilidad otros algoritmos con los cuales se pueda hacer una síntesis correcta de las funciones lógicas que se deseen resolver.