1 El algoritmo de descomposición de funciones booleanas DASG

1.1. Introducción

Una función booleana de tamaño n es linealmente separable si se puede encontrar un hiperplano de tamaño $n-1$ que clasifique correctamente los ejemplos con salida "verdadero" y los de salida "falso". Sea una puerta umbral (*threshold gate , TG*) una puerta lógica con n variables de entrada binaria x_i, $i = \{1, ..., n\}$, y una salida binaria, y, para la que existe un conjunto de n pesos y un valor umbral θ todos ellos reales, tal que la salida es cero si la suma ponderada de las entradas por los pesos es menor que el umbral, y uno en caso contrario (ecuación 1.1). De esta forma, se dice que una función booleana es linealmente separable si puede ser implementada por una puerta umbral.

$$y(\mathbf{X}) = \begin{cases} 0 & si \sum_{i=1}^{n} w_i x_i \leq \theta \\ 1 & en\, caso\, contrario \end{cases} \tag{1.1}$$

Por desgracia, no todas las funciones booleanas son linealmente separables. De hecho, aunque el conjunto de funciones linealmente separables es grande, la gran mayoría de funciones booleanas no lo son (Tab. 1.1). Una función linealmente no separable, por lo tanto, es una función que no puede implementarse con una puerta umbral. Este problema, conocido como "Lógica umbral para la síntesis de funciones booleanas", ha sido ampliamente estudiado desde los años 60 [**?**, **?**, **?**, **?**, **?**]. Esto, que a priori es un problema de difícil solución, se puede resolver generando redes de puertas umbrales, que implementan cada una de ellas funciones linealmente separables y en su conjunto generan la función linealmente no separable deseada.

DASG es un algoritmo de descomposición de funciones booleanas linealmente no separables en funciones booleanas linealmente separables. En el área de diseño de

Número de entradas	Funciones booleanas	FLS	Porcentaje FLS
2	$16 = 2^{2^2} = 2^4$	$14 < 2^4$	$87.5\,\%$
3	$256 = 2^{2^3} = 2^8$	$104 < 2^7$	$40.625\,\%$
4	$65536 = 2^{16}$	$1882 < 2^{11}$	$2.872\,\%$
5	$4294967296 = 2^{32}$	$94372 < 2^{17}$	$0.00002\,\%$
6	2^{64}	$15028134 < 2^{24}$	$< 2^{-40}\,\%$
7	2^{128}	$8378070864 < 2^{33}$	$< 2^{-95}\,\%$
8	2^{256}	$17561539552946 < 2^{44}$	$< 2^{-212}\,\%$
9	2^{512}	$144130531453121108 < 2^{58}$	$< 2^{-454}\,\%$
...	...	...	$\simeq 0\,\%$

Tabla 1.1: Número de funciones booleanas linealmente separables (FLS) frente al conjunto de todas las funciones booleanas para un número de entradas n

circuitos existe un gran interés por la lógica umbral. El desarrollo de la lógica umbral comenzó a principios de 1960 como una teoría global del álgebra de conmutación, incluyendo la lógica booleana convencional como un subconjunto de la lógica umbral. Durante muchos años, el diseño digital basado en puertas umbral se consideró una alternativa al tradicional con puertas lógicas. La potencia del estilo basado en puertas umbral reside en la complejidad intrínseca de las funciones realizadas con dichas puertas, lo que permite que los sistemas implementados contengan menos puertas umbral o menos niveles de puertas que los que tendrían si se realizaran con puertas lógicas tradicionales, y por lo tanto se necesita una menor área de integración para implementar la función [**?**, **?**, **?**]. En particular, puede demostrarse teóricamente que funciones importantes tales como la adición múltiple, la multiplicación, la división o la ordenación, cuya implementación requiere redes de puertas tradicionales en las que el número de niveles no está acotado polinómicamente, pueden implementarse con redes de puertas umbral cuyo número de niveles sí lo está. Las redes de puertas umbrales son útiles también en el modelado de redes nerviosas, para el modelado de sistemas de aprendizaje, de reconocimiento de patrones, de redes neuronales, etc.

Muchas de las características de estos circuitos pueden ser extendidas y aplicadas a las redes neuronales artificiales [**?**, **?**, **?**]. La principal cualidad del algoritmo DASG es su carácter constructivo. DASG genera redes similares a las redes *"feedforward"*, compuestas por una capa de entrada, una capa oculta y una capa de salida. En la capa de salida el algoritmo siempre tendrá implementado el operador lógico AND o el operador lógico OR según necesidad.

La práctica habitual para la selección de arquitecturas en el campo de redes neuronales artificiales es el método de "prueba y error", ya que hasta el momento no existe un forma de conocer ésta a priori. Este es un método muy poco eficiente, por lo que se han propuesto algunos estimadores o heurísticos para intuir cual podría ser una arquitectura apropiada [**?**]. Distintos algoritmos de redes neuronales constructivos han sido propuestos en los últimos años para evitar los inconvenientes del proceso de selección de la arquitectura [**?**, **?**, **?**, **?**] y una comparación y evaluación de esos algoritmos puede ser encontrada en [**?**, **?**]. DASG tiene la ventaja de que se puede extraer conocimiento de el, es decir, una vez entrenada la red se pueden extraer reglas potencialmente interpretables a posteriori por un experto humano. En el **??** se hace uso de esta propiedad obteniendo dicho conocimiento en forma de reglas de un problema real tan importante como puede ser la predicción del mejor tratamiento que se le ha de aplicar a un paciente que ha sido operado de cáncer de pecho para que no se produzca recidiva.

La organización del presente capítulo es la siguiente: En la sec. 1.2 se presentan los conceptos matemáticos necesarios para introducir el algoritmo, en la sec. 1.3 se introducen los detalles del funcionamiento del algoritmo; en la sec. 1.4 se muestran los resultados obtenidos por DASG en la síntesis de funciones booleanas; en la sec. 1.5 se desarrolla una extensión del método para generalización de funciones booleanas, y finalmente en la sec. 1.6 se comentan las conclusiones del trabajo realizado.

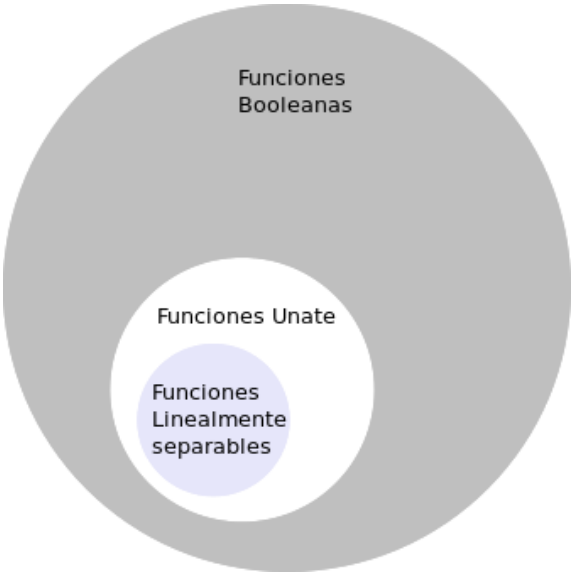

Figura 1.1: Diagrama representativo de la relación de inclusión de las funciones "*Unate*" frente al número de funciones booleanas y las funciones linealmente separables.

1.2. Preliminares matemáticos

Con objeto de poder entender mejor la descomposición aplicada a una función booleana, se presentan a continuación algunos conceptos matemáticos necesarios:

Función booleana: Una función booleana de n variables está definida como una función cuyo dominio son cadenas de tamaño n formadas por valores binarios 0 o 1, y cuyo codominio son ambos valores 0 y 1. Formalmente, son funciones de la forma:

$$f : B^n \to B \tag{1.2}$$

donde $B = \{0, 1\}$ y $n \in \mathbb{N}$.

Función booleana parcialmente definida: Una función booleana parcialmente definida de n variables está definida como una función cuyo dominio son cadenas de tamaño n formadas por valores binarios 0 o 1, y cuyo codominio son ambos

valores 0 y 1, y un valor adicional * para representar a los valores no conocidos. Formalmente, son funciones de la forma:

$$f : B^n \rightarrow B^* \tag{1.3}$$

donde $B = \{0, 1\}$, $B^* = \{0, 1, *\}$ y $n \in \mathbb{N}$.

Vector de verdad: Una función booleana está completamente especificada por la salida de esta, $f(\mathbf{x})$, como respuesta de cada una de sus 2^n posibles entradas, pudiendo ser representada por un vector $t = [t_o, t_1, \cdots, t_{2^n-1}]$ donde $t_i = f(x_i)$, $x_i = bool(i, n)$, y $bool(k, j) = valor\, booleano\, K - ésimo\, de\, entradas\, de\, tamaño\, j$.

Puerta umbral: Puerta lógica con n variables de entrada binaria $x_i, \{i = 1, ..., n\}$, y salida binaria, y, para la que existe un conjunto de $n + 1$ números reales, el umbral θ, y los n pesos w_i, tal que la salida es cero si la suma ponderada de las entradas $(\sum w_i x_i)$ es menor que el umbral, y uno en caso contrario. La función computada por una puerta umbral de n variables es una función booleana f de n variables tal que para cada entrada $\mathbf{X} = (x_0, x_1, \ldots, x_{n-1})$, el valor de la función es:

$$f(\mathbf{X}) = \begin{cases} 1 & si\ \sum_{i=0}^{n-1} w_i x_i - \theta \geq 0 \\ 0 & en\, otro\, caso \end{cases} \tag{1.4}$$

Función linealmente separable: Cualquier función que puede ser computada por una puerta umbral se denomina función umbral o función linealmente separable. Se dice que una función de n variables es linealmente separable cuando existe un hiperplano de tamaño $n - 1$ que separa los ejemplos con salida "0" de los de salida "1".

Peso Hamming: El peso Hamming H de una función booleana se define como el número de valores "1" que contiene el vector de la verdad.

Función simétrica: Una función simétrica es una función que su vector de la verdad depende solo del peso Hamming de la entrada, es decir, cualquier permutación de las variables de la entrada no produce ningún cambio en la salida.

Negación de una variable: La negación de una variable x_i (escrita como $\bar{x_i}$) consiste en cambiar su valor de 0 a 1 o viceversa ($\bar{x_i} = 1 - x_i$).

Función positiva/negativa: Una función $f(x_0, x_1, \ldots, x_{n-1})$ es positiva en x_i si y solo si $f(x_0, x_1, \ldots, x_i = 0, x_{n-1}) \leq f(x_0, x_1, \ldots, x_i = 1, x_{n-1})$. De igual manera, una función $f(x_0, x_1, \ldots, x_{n-1})$ es negativa en x_i si y solo si $f(x_0, x_1, \ldots, x_i = 0, x_{n-1}) \geq f(x_0, x_1, \ldots, x_i = 1, x_{n-1})$, al igual que en el caso anterior.

Función Unate en una variable: Una función booleana se dice que es $Unate$ en una variable x_i, sí y solo sí $f(x_0, x_1, \ldots, x_{n-1})$ es positiva/negativa en x_i.

Función Unate: Si una función booleana es $Unate$ para todas sus variables, se dice que es $Unate$. Cada función booleana linealmente separable es $Unate$, pero no todas las funciones $Unate$ son linealmente separables (ver Fig. 1.1).

Influencia de una variable (sensitividad): La influencia de una variable mide el número de veces respecto a todas las entradas posibles posibles, cuando un cambio en una de las variables de la entrada manteniendo el resto de la variables invariantes, produce un cambio en la salida de la función. La influencia de una variable x_i está definida como 0,5 veces el número de vectores de entrada $x \in \{0,1\}^n$ tales que la negación de la variable valor i-esima de la entrada cambia el valor de la salida de la función. Para clarificar, si un vector de entrada $\mathbf{X} = (x_0, x_1, \ldots, x_i, \ldots, x_{n-1})$ es modificado negando la variable x_i, obteniendo el vector $\mathbf{X}_{\bar{x_i}} = (x_0, x_1, \ldots, \bar{x_i}, \ldots, x_{n-1})$ y la salida de la función booleana f

varia para estas dos entradas ($f(\mathbf{X}) \neq f(\mathbf{X}_{\bar{x_i}})$), entonces la influencia de la variable x_i se incrementará en 1. Un ejemplo del calculo de la influencia puede ser encontrado en el sec. 1.3.5.

Extensión autodual de una función booleana: La extensión autodual de una función booleana de n variables es la función en $n+1$ dimensiones y se definida como:

$$f^{n+1}(x_0, \ldots, x_{n-1}, x_n) = \begin{cases} f^n(x_0, \ldots, x_{n-1}) & si\, x_n = 0 \\ 1 - f^n(\bar{x}_0, \ldots, \bar{x}_{n-1}) & si\, x_n = 1 \end{cases} \tag{1.5}$$

Fan-in: Número máximo de entradas de una puerta lógica

1.3. Algoritmo DASG

El algoritmo de aprendizaje DASG genera arquitecturas de redes tipo $feedforward$ de una única capa oculta compuesta por neuronas umbrales y una neurona en la capa de salida.

La Fig. 1.2 muestra un ejemplo de una arquitectura de red generada por el algoritmo una vez terminado el entrenamiento. En este caso, la figura muestra una arquitectura generada con una puerta OR como neurona de salida, aunque dependiendo de la función objetivo a descomponer, se podría haber usado una puerta AND como neurona de salida. Cada una de las M puertas umbrales de la capa oculta han sido obtenidas descomponiendo la función objetivo en funciones linealmente separables más simples, tales que su unión por medio de los operadores lógicos OR o AND en

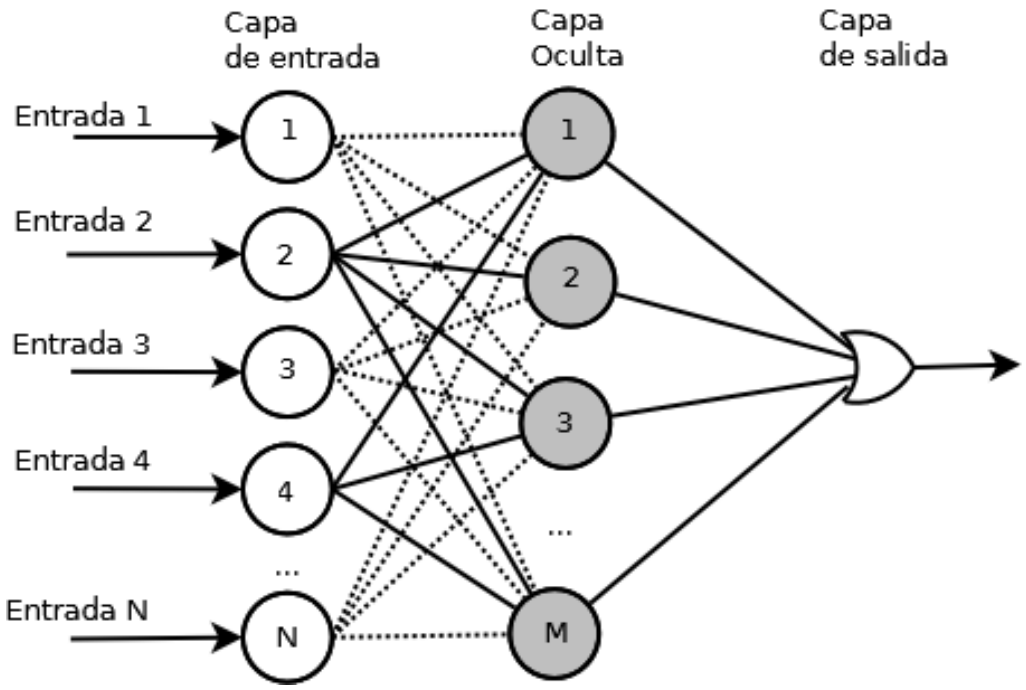

Figura 1.2: Arquitectura de una red DASG generada con una puerta OR de salida.

la capa de salida, generan de nuevo la función objetivo (1.6).

$$f_{obj}\left(x_1, \cdots, x_n\right) = OR/AND(f_1\left(x_1, \cdots, x_n\right), \cdots, f_m\left(x_1, \cdots, x_n\right)) \tag{1.6}$$

$f_i\left(x_1, \cdots, x_n\right)$ *es linealmente separable*, $\forall i \in \{1, ...m\}$

$f_{obj}\left(x_1, \cdots, x_n\right)$ *no es linealmente separable*

Por lo tanto, la idea del algoritmo es reducir iterativamente la complejidad de la fun-

ción objetivo dividiéndola en cada paso en dos funciones más simples tales que su conjunción o disyunción generé nuevamente la función objetivo. En caso de que las dos nuevas funciones booleanas obtenidas de la división anterior no sean linealmente separables, se usará nuevamente la misma técnica hasta que todas las funciones booleanas resultantes lo sean. DASG almacena en memoria dos conjuntos de funciones booleanas: *workSet* (funciones a analizar), que almacena todas las funciones que aún no han sido analizadas su separabilidad lineal, y *SolutionSet*, que almacena todas las funciones linealmente separables encontradas hasta el momento, de manera que cada función del conjunto *SolutionSet* será implementada por una puerta umbral en la capa oculta en la arquitectura final de la red. DASG inicialmente empieza con arquitectura sin ninguna neurona en la capa oculta, una neurona de salida implementando la puerta umbral AND o OR según necesidad, y únicamente la función objetivo almacenada en *workSet* (ver Fig. 1.3).

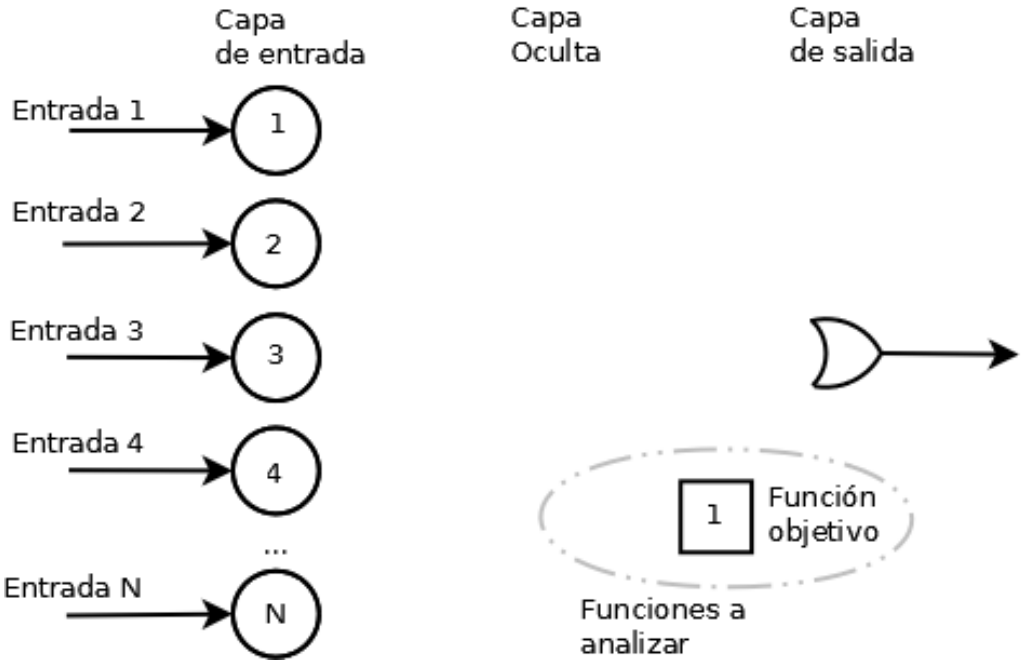

Figura 1.3: Arquitectura inicial de una red DASG generada con una puerta OR de salida.

Este proceso de descomposición es incremental e iterativo; en cada iteración la red

selecciona una función del conjunto de funciones a analizar (*workSet*) y comprueba su separabilidad lineal, añadiéndola a la capa oculta en caso de que lo fuera, o descomponiéndola en dos nuevas funciones más simples que serán añadidas al conjunto de funciones a analizar en caso contrario. La Fig. 1.4 muestra una iteración del algoritmo en la cual K puertas umbrales ya han sido encontradas y quedan por analizar todavía E funciones que podrían ser linealmente separables con lo que se añadirían a la capa oculta, o no serlo, con lo cual tendrían que ser divididas en dos funciones más simples, y así nuevamente hasta que no quedasen funciones que analizar en el conjunto *workSet*.

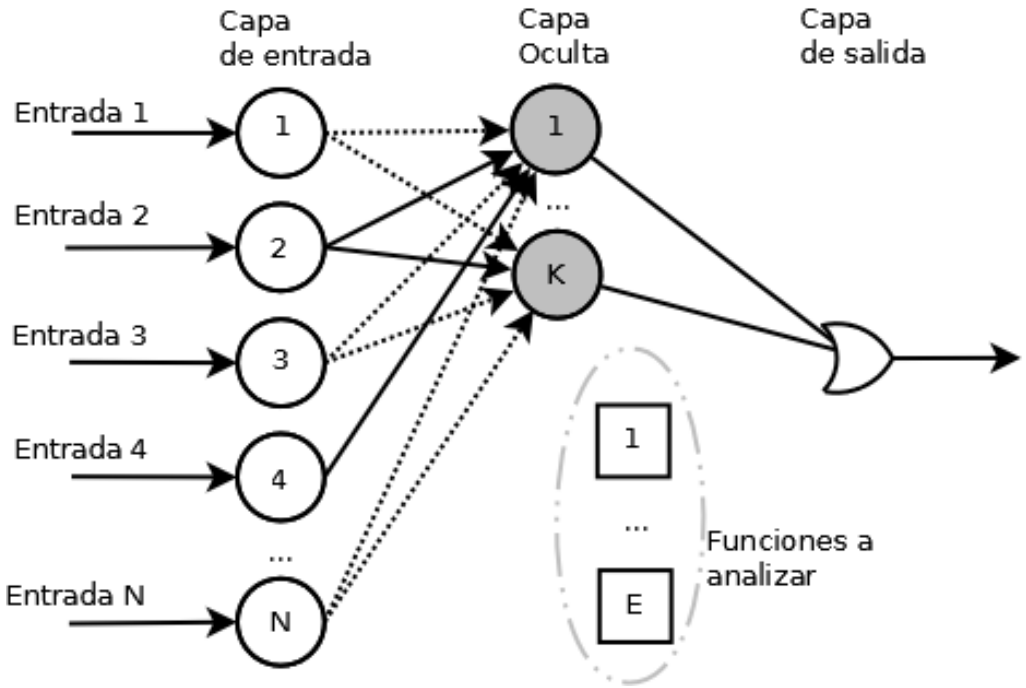

Figura 1.4: Generación de una arquitectura de red DASG utilizando con una puerta OR de salida. En la figura se observa la generación de la capa oculta, en la cual ya se han encontrado K puertas umbrales linealmente separables. El conjunto de datos de funciones a analizar son todas aquellas funciones para las que aún no han sido analizadas su separabilidad lineal. Cuando no queden funciones en este conjunto, el algoritmo de aprendizaje habrá finalizado.

La Fig. 1.5 nos muestra un diagrama de flujo del algoritmo de aprendizaje en detalle. La función objetivo es el único argumento de entrada del algoritmo y será almace-

nada inicialmente en el conjunto *workSet*. Es en este momento donde se decide cual será la función que se utilizará en la capa de salida: una función AND o una función OR. La neurona de salida implementará una función $OR(AND)$ si más de la mitad de las salidas de la función objetivo son $true(false)$. Esta elección no es arbitraria y esta motivada por los resultados obtenidos en [**?**], donde se comprobó que existía una fuerte correlación entre el porcentaje de ejemplos con salida activa de la función objetivo y el porcentaje de ejemplos con salida activa de la función implementada por la neurona de salida de la red, de manera que existen muchas más configuraciones posibles de pesos sinápticos en las neuronas de la capa oculta que logran que la red implemente una función objetivo con muchos (pocos) ejemplos activos si utilizamos una neurona en la capa de salida que implemente una función con muchos (pocos) ejemplos activos . Hay que decir que si la función objetivo esta balanceada es irrelevante usar una función de salida AND o una función de salida OR.

DASG continua ejecutándose mientras existan elementos en la lista *workSet*. En cada iteración extrae una de las funciones del conjunto y analiza su separabilidad lineal. Si la función es linealmente separable (LS) se añade al conjunto *SolutionSet*. En caso contrario, la función se divide en dos funciones más simples que se añaden a la lista *workSet* para ser analizadas con posterioridad. La división de la función en términos de una variable se conoce como descomposición de Shannon, y fue introducida en el año 1938. Los algoritmos de descomposición son un tópico clásico de la investigación de la teoría de la computación, y han sido ampliamente estudiados durante la década de los 60 [**?**, **?**, **?**, **?**, **?**].

Analizar la separabilidad lineal de una función dada no es un problema trivial. Para facilitar esta tarea, el algoritmo primero analiza si la función es *unate*. Si una función es *unate*, no implica que la función sea linealmente separable, pero si una función

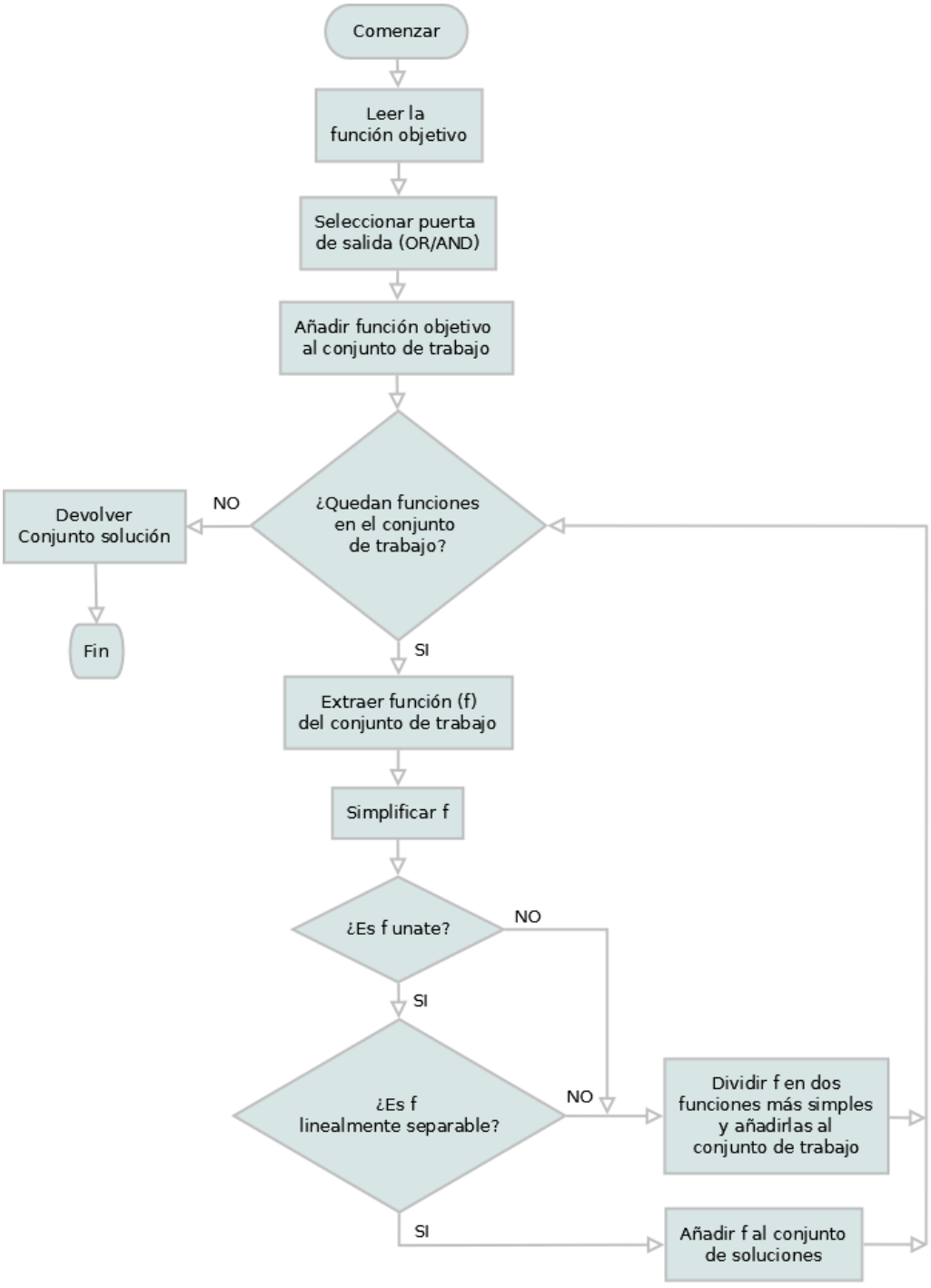

Figura 1.5: Diagrama de flujo del algoritmo de aprendizaje DASG donde se descompone una función objetivo dada en un conjunto de funciones linealmente separables (LS).

es linealmente separable siempre será *unate*. En este caso, analizamos si la función es *unate* en su espacio autodual ya que ser *unate* en el espacio autodual $n+1$ es un filtro más restrictivo que ser *unate* en el espacio n. ([**?**], Teorema 3.1). Por lo tanto, una forma rápida de identificar funciones LNS es cuando una función no es *unate* en el espacio autodual, pero si la función es *unate*, no implica que que la función sea LS, ya que existen funciones booleanas que no son linealmente separables pero si son *unate* en su espacio autodual. Esto implica que hay que comprobar su separabilidad lineal con una segunda prueba usando un algoritmo de separabilidad lineal. El algoritmo usado en este segundo test fue *simplex* [**?**, **?**]. El motivo de no usar directamente al algoritmo *simplex* para comprobar su separabilidad es que comprobar en primer lugar la propiedad de "unicidad" es sensiblemente más rápido en tiempo de ejecución.

1.3.1. Importancia del peso de la función objetivo en la selección de la función de salida

El algoritmo DASG puede funcionar con una neurona de salida AND o una OR de acuerdo a la función objetivo. En [**?**] se analizó la capacidad computacional de redes neuronales $feed-forward$ para generar funciones booleanas con un tamaño de entrada $N=3$. En concreto, haciendo uso del algoritmo desarrollado en [**?**], se hizo un estudio exhaustivo en el que se analizó la función de salida más adecuada para poder resolver cada una de las 152 funciones booleanas linealmente no separables de tres entradas. DASG usa como función de salida una función AND o una OR en base al peso Hamming de la función objetivo. Se han hecho simulaciones del algoritmo con funciones de salida AND y OR con la misma función objetivo, y los resultados mostraron que se generaban arquitecturas más pequeñas cuando la función de salida seleccionada seguía los criterios anteriormente expuestos.

1.3.2. Proceso de simplificación de una función

Se ha incorporado al algoritmo un proceso de simplificación que si bien no es necesario, es muy importante ya que reduce el coste computacional del algoritmo al reducirse el tamaño de entrada de las funciones a analizar. Una función booleana puede ser simplificada eliminando una variable x_i si la influencia de la variable i-ésima es cero. Esto es trivial, ya que si la influencia de la variable i-ésima es cero es porque no existe ningún efecto de la variable en la salida de la función, es decir, cambios de la variable en la entrada no afectan en nada a la salida de la función. Para simplificar por una variable x_i una función f de n entradas, obteniéndose una función f' de $n-1$ entradas, se aplica la siguiente ecuación:

$$f_{simplificada}(x_0, x_1, ..., x_{i-1}, x_{i+1}, ..., x_{n-1}, x_n) = \\ f_{original}(x_0, x_1, ..., x_{i-1}, 0, x_{i+1}, ..., x_{n-1}, x_n)$$

En la Tab. 1.2 se muestra un ejemplo de simplificación de una variable en una función lógica de tres variables (Figura 1.1a). En este ejemplo se muestra una función en la que la variable x_0 tiene una sensitividad de cero, pudiendo ser eliminada sin perder ninguna información usando la ecuación $f_{simplificada}(x_1, x_2) = f_{original}(0, x_1, x_2)$.

1.3.3. Test de separabilidad lineal

Un punto fundamental del algoritmo consiste en analizar la separabilidad lineal de una determinada función booleana, siendo deseable que el método elegido para realizar este análisis fuera lo más rápido y eficiente posible. Todas las funciones umbrales son *unate*, por lo que el primer paso a realizar es comprobar la "unicidad" de la función. L. Franco et al. [**?**] han mejorado la comprobación de la unicidad de una función demostrando que analizar la unicidad en el espacio autodual de

(a) Función lógica de tres entradas que se desea simplificar.

x_0	x_1	x_2	$f_{original}(x_0, x_1, x_2)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b) Función lógica resultante de haber eliminado la variable x_0.

x_1	x_2	$f_{simplificada}(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 1.2: Ejemplo de simplificación de una función lógica de tres variables. La variable x_0 tiene una influencia igual a cero, por lo que eliminamos esa variable, generándose una función equivalente con un número menor de entradas .

$n + 1$ variables es una prueba mas restrictiva que chequearlo en el espacio original de n dimensiones. Sin embargo, existen funciones *unates* que no son linealmente separables, y con intención de detectar aquellas funciones no linealmente separables que si son *unate*, se aplica una segunda prueba. La prueba esta basada en un método de separabilidad lineal entero conocido como *simplex para la que se* ha usado una versión publica obtenida del paquete $LPSolve$ [**?**, **?**], que permite obtener los valores de los pesos sinápticos y umbral mínimos para resolver la función en caso de que esta sea LS. Estos pesos y umbrales son los usados en la generación de la puerta umbral que implementa la función.

1.3.4. Descomposición basada en la influencia de la variable

Cuando el algoritmo de aprendizaje encuentra un función no linealmente separable (F^k), esta se descompone usando la variable *no-unate* con la influencia más alta. Esta forma de división genera dos funciones, F_b^{k+1} y F_b^{k+1}, que se obtienen de acuerdo a las dos siguientes ecuaciones:

$$F_a^{k+1}(x_0,\ldots,x_i,\ldots,x_{n-1}) = \begin{cases} F^k(x_0,\ldots,x_i=0,\ldots,x_{n-1}) & si\, x_i = 0 \\ t & si\, x_i = 1 \end{cases} \tag{1.7}$$

$$F_b^{k+1}(x_0,\ldots,x_i,\ldots,x_{n-1}) = \begin{cases} t & si\, x_i = 0 \\ F^k(x_0,\ldots,x_i=1,\ldots,x_{n-1}) & si\, x_i = 1 \end{cases} \tag{1.8}$$

donde

$$t = \begin{cases} 0 & si\, función\, de\, salida = OR \\ 1 & si\, función\, de\, salida = AND \end{cases}$$

El proceso de descomposición genera 2 funciones más simples, ya que cada una de las dos funciones obtenidas tienen asegurada su unicidad (positiva o negativa) para la variable x_i por la que fue descompuesta la función original. Supongamos que tenemos una función $f(x_0, x_1, \ldots, x_{n-1})$ que es *no-unate* en una variable x_i. Esto quiere decir que $f(x_0, x_1, \ldots, x_{n-1})$ no es ni positiva, ni negativa en x_i por que existen vectores de entradas $\mathbf{X}_{x_i=0}$, $\mathbf{X}_{x_i=1}$,$\mathbf{Y}_{x_i=0}$, $\mathbf{Y}_{x_i=1}$ donde $f(\mathbf{X}_{x_i=0}) < f(\mathbf{X}_{x_i=1})$

y $f(\mathbf{Y}_{x_i=0}) > f(\mathbf{Y}_{x_i=1})$. El proceso de descomposición de la función esta pensado para que $F_a(x_0,\ldots,x_i,\ldots,x_{n-1})$ sea positiva (negativa) para la variable x_i, y $F_b(x_0,\ldots,x_i,\ldots,x_{n-1})$ sea negativa (positiva) para la variable x_i si la función de salida es una *AND* (*OR)*. Para demostrarlo, se supone que la función de salida es una función *AND,* por lo que al descomponer la función f en las funciones F_a y F_b, las salidas de las funciones para las entradas $\mathbf{X}_{x_i=0}$, $\mathbf{X}_{x_i=1}$,$\mathbf{Y}_{x_i=0}$, $\mathbf{Y}_{x_i=1}$ serán:

$$F_a(\mathbf{X}_{x_i=0}) = f(\mathbf{X}_{x_{i=0}})\, y\, F_a(\mathbf{X}_{x_i=1}) = 1$$
$$F_b(\mathbf{X}_{x_i=0}) = 1\, y\, F_b(\mathbf{X}_{x_i=1}) = f(\mathbf{X}_{x_i=1})$$
$$F_a(\mathbf{Y}_{x_i=0}) = 1\, y\, F_a(\mathbf{Y}_{x_i=1}) = f(\mathbf{Y}_{x_{i=0}})$$
$$F_b(\mathbf{Y}_{x_i=0}) = f(\mathbf{Y}_{x_{i=0}})\, y\, F_b(\mathbf{Y}_{x_i=1}) = 1$$

por lo que $F_a(\mathbf{X}_{x_i=0}) \leq F_a(\mathbf{X}_{x_i=1})$, $F_a(\mathbf{Y}_{x_i=0}) \leq F_a(\mathbf{Y}_{x_i=1})$, y $F_b(\mathbf{X}_{x_i=0}) \geq F_b(\mathbf{X}_{x_i=1})$, $F_b(\mathbf{Y}_{x_i=0}) \geq F_b(\mathbf{Y}_{x_i=1})$, y por lo tanto son *unates* para la variable x_i.

Se probaron diferentes estrategias para la elección de la variable con la cual dividir una función no linealmente separable, encontrando que era mas eficiente seleccionar aquella *no-unate* que tenía un sensitividad más alta. Las otras dos estrategias que se analizaron fueron: i) Seleccionar una variable aleatoriamente, con la que se incrementó el número de puertas necesarias en un 46.1 % de media, y en un 63,7 % el nivel de la interconexión. ii) Dividir por la variable *no-unate* con un mayor valor de influencia positiva y negativa. En este caso el número de puertas generadas se incrementó en un 16,1 % mientras que la interconexión en un 19,2 %.

En el caso de caso de funciones que son *unate* en el espacio autodual $n+1$, pero que no son LS, la selección de la variable se realiza eligiendo al criterio de mayor influencia.

1.3.5. Ejemplo de la aplicación del algoritmo a una función booleana de cuatro variables

A continuación se analiza el algoritmo de descomposición para el caso de una función de cuatro entradas. La función objetivo tiene como vector de verdad el vector $\mathbf{V} = \{1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1\}$ (Tab. 1.3). Una representación de la función por sumas de productos en lógica booleana seria $f(x_0, x_1, x_2, x_3) = x_0x_3 + x_0x_1\bar{x}_3 + x_0\bar{x}_1\bar{x}_2 + \bar{x}_0x_2\bar{x}_3 + \bar{x}_0x_1\bar{x}_2x_3$. La función no es *unate,* y por lo tanto no es LS. El procedimiento de descomposición general de esta función se muestra en la Fig. 1.6. En la parte superior de la figura se muestra el vector de verdad de la función objetivo. El peso de la función objetivo es 11, con lo que más de la mitad de los ejemplos son 1, por lo tanto la función implementada en la capa de salida será una OR.

x_0	x_1	x_2	x_3	$Salida$
0	0	0	0	1
1	0	0	0	0
0	1	0	0	0
1	1	0	0	1
0	0	1	0	1
1	0	1	0	0
0	1	1	0	1
1	1	1	0	1
0	0	0	1	1
1	0	0	1	1
0	1	0	1	1
1	1	0	1	1
0	0	1	1	0
1	0	1	1	1
0	1	1	1	0
1	1	1	1	1

Tabla 1.3: Tabla de verdad de la función booleana con vector de verdad {1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1}. Se puede observar que el vector de la verdad de la función coincide con la columna salida de la tabla de verdad.

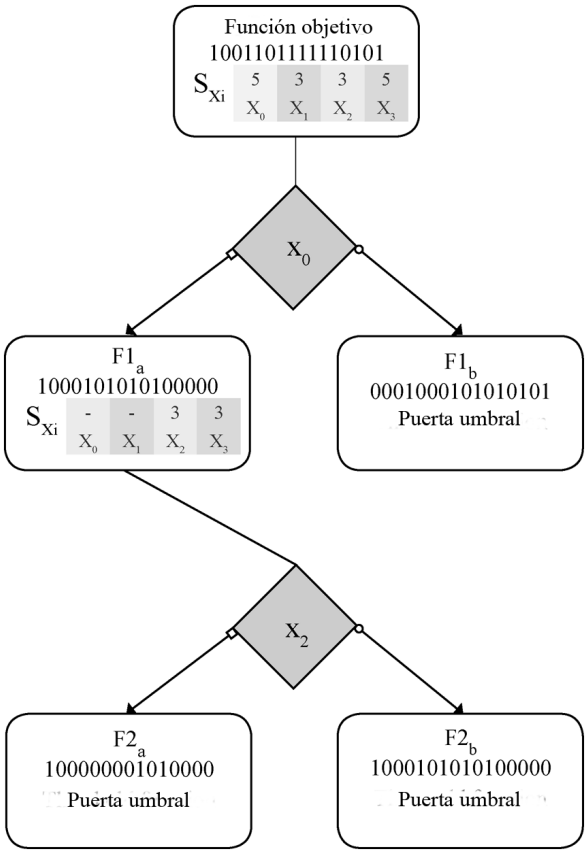

Figura 1.6: Ejemplo de la aplicación del algoritmo DASG a una función booleana con vector de la verdad {1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1}.

El algoritmo irá descomponiendo iterativamente la función mientras las funciones sean LNS. Cada descomposición se ve representada como un rombo en donde se indica qué variable ha sido elegida para realizar la división. Definimos S_{x_i} como la influencia de la variable x_i, siendo x_i *nounate*. Por ejemplo, el valor de la influencia para la variable x_1 en la primera división será el número de veces que cambiando el valor de la entrada x_1 de 0 a 1 o viceversa, el valor de la salida también cambia.

En la Tab. 1.3 se puede observar, por ejemplo, que para los vectores de entradas $\mathbf{X}_1$={1, 0, 0} y $\mathbf{X}_2$={1,1,0}, donde solo ha cambiado la entrada x_1, existe un cambio en la salida de 0 a 1. Por lo tanto, analizando todas las posible entradas, el valor de la influencia de x_1 es de $S_{xi} = 0,5 \times 6 = 3$.

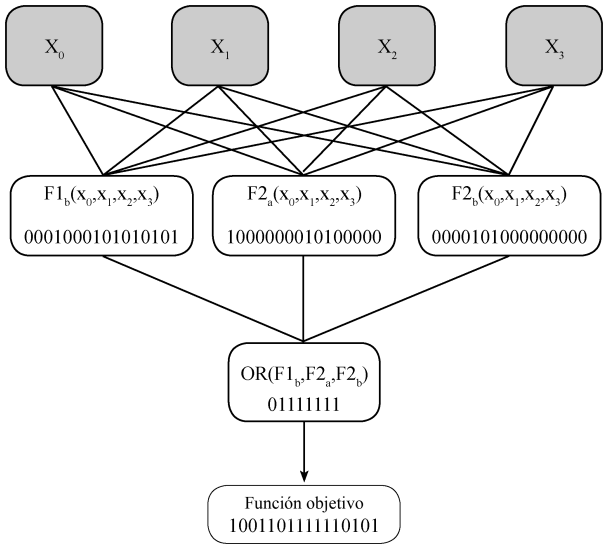

Figura 1.7: Solución final encontrada por el algoritmo DASG para la función de cuatro variables con un vector de verdad {1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1} mostrado en la Tab. 1.3. El procedimiento usado para obtener la solución se encuentra descrito en el sec. 1.3.5 y se muestra en la Fig. 1.6.

En cada caja de la Fig. 1.6 se muestra cada una de las influencias para cada una de las cuatro variables de la función que se está analizando en ese momento. La influencia de aquellas variables que sí son *unate* no se muestran con objeto de mostrar que esas variables no se tienen en cuenta para la división de la función.

En la caja superior de la Fig. 1.6 se muestran las influencias de las cuatro variables de la función objetivo inicial. Las variables x_0 y x_3 tienen una influencia de 5, siendo el valor más alto de todas las variables *no-unates.* En este caso, cualquiera de las dos variables puede ser seleccionada para dividir la función, siendo x_0 la variable elegida en este caso. Para dividir la función se usan las ecuaciones 1.7 y 1.8, obteniéndose dos nuevas funciones F_a^1 y F_b^1. Estas funciones heredarán la mitad de los bits originales de la función F de las que fueron obtenidas mientras que el resto de los bits serán 0 dado que se utiliza una función de salida OR.

El procedimiento continúa chequeando si las nuevas funciones son LS. En el ejemplo una de las funciones creadas (F_b^1) es una función LS, por lo que es añadida

directamente al conjunto solución. La función F_a^1 no es linealmente separable por lo que tendrá que ser nuevamente descompuesta en otras dos funciones, F_a^2 y F_b^2 tras dividirla por la variable x_2. Finalmente, las dos nuevas funciones F_a^2 y F_b^2 sí son linealmente separables y por lo tanto se añaden al conjunto solución. Una vez finalizada la descomposición, al no quedar más funciones que analizar, se utilizará todo el conjunto solución para generar la arquitectura de red final mostrada en la Fig. 1.7.

1.4. Síntesis de funciones booleanas usando DASG

En este apartado se analiza la capacidad del algoritmo DASG para producir redes compactas con muy pocas puertas umbrales y una baja interconexión, con un conjunto de funciones lógicas de circuitos muy utilizado para pruebas de síntesis de funciones lógicas (MCNC[1]) y utilizando un número variable de entradas entre 3 y 257. Debido a que los circuitos electrónicos tienen un *fan-in* limitado, se analizá el rendimiento de este algoritmo en 2 etapas. En el sec. 1.4.1 se muestra la aplicación del algoritmo a la tabla de verdad de las funciones objetivo de menos de 21 entradas, mientras que en el sec. 1.4.2, se aplica el algoritmo DASG tras haber preprocesado la función con el programa SIS para reducir el *fan-in* a un máximo de 21.

Se han comparado los resultados obtenidos por DASG con los obtenidos por Zhang [**?**], ya que este algoritmo obtuvo un gran mejora respecto a otros algoritmos previamente publicados.

[1] http://cadlab.cs.ucla.edu/ %7Ekirill/blif-benchmarks.zip

1.4.1. Síntesis de funciones booleanas de hasta 21 entradas

En la Tab. 1.4 se muestran las principales características de las arquitecturas obtenidas por el algoritmo DASG: el número total de puertas umbrales usadas (G), el número de niveles o capas ocultas usadas (L), la interconexión o número de conexiones sinápticas totales de la red (W), y el *fan-in-max* (F). Los resultados son comparados a los obtenidos por Zhang [**?**]. Las tres últimas columnas muestran el porcentaje de mejora (reducción) en el número de puertas, número de conexiones sinápticas, y el número de niveles en comparación con el algoritmo de Zhang, donde los valores negativos indican que los resultados del algoritmo fueron peores.

La aplicación del algoritmo DASG a la definición de la función genera únicamente una sola capa oculta de puertas umbrales (o neuronas). Para conseguir una reducción del *fan-in* de la capa de salida se puede modificar la arquitectura fácilmente incrementando el número de capas. El *fan-in* máximo de los circuitos generados es habitualmente igual al número de variables de entrada pertinentes, es decir, variables que influyen en la salida de la función. No obstante, si el número de funciones umbral generadas en la capa oculta es mayor que el número de variables de entrada de la función, la neurona de salida tendría un *fan-in* igual a este número de puertas. En este caso una transformación muy simple (Fig. 1.8) puede reducir el *fan-in* máximo hasta el número de variables de entrada correspondiente. Esta transformación es posible porque la neurona de salida de las arquitecturas es una OR o una AND.

Para todas las funciones del conjunto de prueba con un número de variables de entrada menor de 11, el tiempo de computo fue inferior a 2 segundos, pero cuando el número de variables aumenta, el coste computacional crece considerablemente debido tanto a la necesidad de potencia de cálculo como a la de recursos de memoria. Por ejemplo, para las tres funciones de salida binaria $pm1c0$, $sctd0$ y $cm150av$ con 16, 19 y 21 variables respectivamente, el tiempo necesario para obtener la solución final

fue de 28, 2382, y 49.383 segundos. Debido esto se limitó el conjunto de funciones analizadas hasta un máximo de 21 variables de entrada.

Se ha excluido de este análisis la función de paridad de 16 variables utilizada en [**?**].

Nombre	I/O	DASG				% reducción		
		G	L	W	F	G	L	W
b1	3/4	7	2	16	3	12	33	0
cm42a	4/10	10	1	40	4	23	67	-18
decod	5/16	16	1	80	5	33	67	-54
cm82a	5/3	11	2	45	5	8	50	-18
majority	5/1	1	1	5	5	0	50	0
z4ml	7/4	16	2	79	7	16	60	-23
f51m	8/8	32	2	153	4	61	75	42
9symml	9/1	21	2	200	20	81	78	51
alu2	10/6	96	2	822	46	51	92	-15
x2	10/7	17	2	79	7	-13	50	-18
cm152a	11/1	9	2	40	8	18	50	5
cm85a	11/3	19	2	180	16	-36	60	-150
cm151a	12/2	18	2	96	8	-50	60	-113
alu4	14/8	279	2	3189	86	32	91	-127
cm162a	14/5	15	2	72	10	42	75	18
cu	14/11	22	2	122	10	8	50	-60
cm163a	16/5	21	2	105	8	16	67	-25
cmb	16/4	71	4	48	12	85	83	32
pm1	16/13	17	2	77	9	26	50	-1
tcon	17/16	24	2	56	2	25	33	0
pcle	19/9	32	2	166	11	8	67	-52
sct	19/15	23	2	134	13	39	60	-16
cm150a	21/1	17	2	112	16	19	50	-45
cc	21/20	44	2	189	6	-26	67	-108
Media		34.91	1.95	254.37	13.37	19.91	61.87	-28.96

Tabla 1.4: Resultados obtenidos en la síntesis de 24 funciones booleanas de varias salidas usando el algoritmo DASG. Las columnas indican el nombre de la función, el número de entradas/salidas de esta, el número de puertas utilizadas (G), el número de niveles (L), las conexiones entre puertas (W) y el *fan-in*-max (F) del circuito generado. Se muestra asimismo (columnas G, L, y W) el porcentaje de mejora obtenido con respecto a los resultados obtenidos en [**?**].

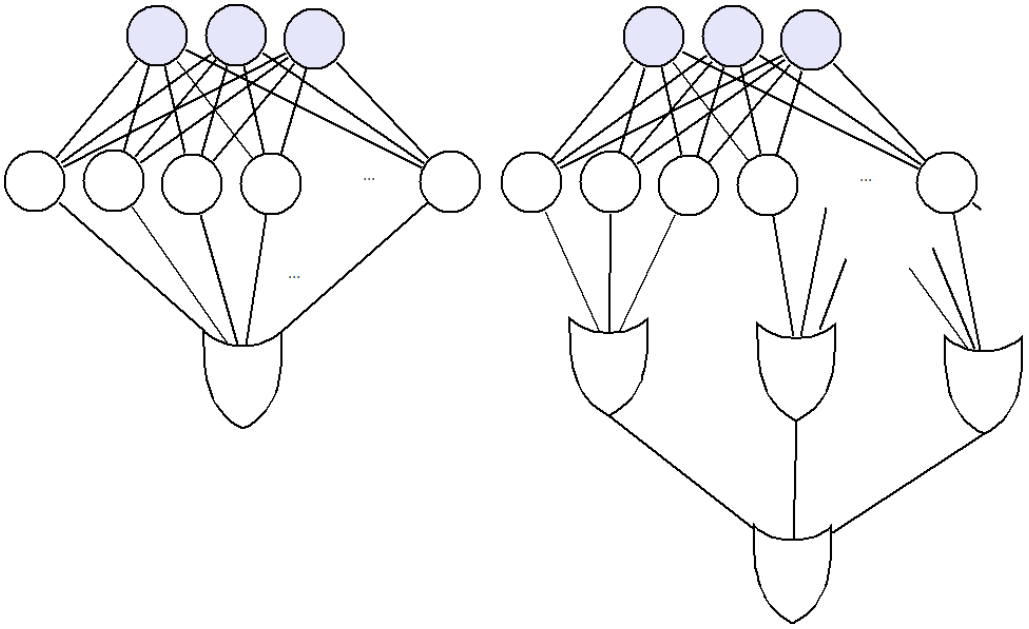

Figura 1.8: Reducción del *fan-in* de una puerta de salida *OR*.

La función de la paridad [**?**, **?**] es una conocida y muy compleja función simétrica y constituye el peor de los casos posibles para este algoritmo, ya que el número de puertas necesarias para su implementación aumenta de forma exponencial (2^{N-1}). La forma en que trabaja el algoritmo DASG hace que sea ineficiente para funciones simétricas. Afortunadamente, el conjunto de funciones de lógicas simétricas es muy pequeño, y comprende sólo una pequeña fracción del conjunto total de funciones lógicas. En la última fila de la Tab. 1.4 se muestra la media de los resultados obtenidos. Se puede ver que el algoritmo DASG supera los resultados obtenidos por Zhang et al. [**?**] en términos del número de puertas y niveles utilizados en el circuito, alcanzando en algunos casos un máximo del 85 %. En promedio se observó una reducción del 19.91 % en el número total de puertas. Para este conjunto de funciones, el algoritmo DASG crea arquitecturas con sólo dos niveles, aunque las arquitecturas generadas tienen un mayor número de conexiones en comparación con los obtenidos por Zhang (un 29 % superior). Este incremento pone de manifiesto un compromiso entre el número de conexiones, el número de puertas y los niveles de

las arquitecturas construidas. Sin embargo, es importante destacar que en 5 de las 24 funciones analizadas ($f51m$, $9symml$, $cm152a$, $cm162a$ y CMB) se obtuvo una reducción simultanea en todos los aspectos analizados.

Nombre	I/O	DASG				% reducción		
		G	L	W	F	G	L	W
cordic	23/2	33	7	130	8	33	-17	16
ttt2	24/21	77	2	462	14	23	67	-264
i1	25/16	21	3	69	7	9	40	-10
la1	26/19	51	3	186	8	6	57	-11
pcler8	27/17	40	3	124	13	15	57	13
frg1	28/3	59	5	207	9	0	44	11
c8	28/18	67	2	216	13	21	71	5
compl	32/3	52	5	208	7	37	38	33
my adder	33/17	89	9	275	6	7	50	10
term1	34/10	90	6	318	8	60	40	53
count	35/16	57	5	238	9	28	58	1
unreg	36/16	64	2	208	4	-28	60	-55
cht	47/36	110	3	352	5	32	40	-74
apex7	49/37	126	6	496	4	42	33	-12
x1	51/35	138	4	501	4	8	43	31
dalu	75/16	571	7	2421	8	30	70	6
example2	85/66	186	6	520	7	-2	25	-6
i9	88/63	437	7	1242	6	-59	13	-52
x4	94/71	216	7	729	12	-14	13	-30
i3	132/6	102	3	166	8	35	50	30
i5	133/66	82	4	134	7	-24	33	-20
i8	133/81	486	5	112	6	15	50	-1
apex6	135/99	438	6	189	12	-11	50	-58
x3	135/99	448	8	77	5	-2	-14	13
i6	138/67	267	2	56	4	3	60	-43
pair	173/137	841	10	166	12	7	17	-12
i4	192/6	74	3	464	15	0	40	-38
i7	199/67	327	2	112	5	-8	60	-46
i2	201/1	68	4	189	9	66	43	60
des	256/245	2398	11	189	8	-25	31	-55
i10	257/224	1511	21	189	8	17	40	19
Media		307.3	5.5	353	8.1	10.35	40.7	-15.7

Tabla 1.5: Resultados obtenidos en la síntesis de 24 funciones booleanas usando el algoritmo DASG. Las columnas indican el nombre de la función, el número de entradas/salidas de ésta, el número de puertas utilizadas (G), el número de niveles (L), las conexiones entre puertas (W), y el *fan-in* máximo (F) del circuito generado. En las tres las últimas columnas se muestra el porcentaje de mejora obtenido con respecto a los resultados publicados por Zhang.

1.4.2. Síntesis de funciones booleanas con un número de entradas superior a 21

La versión implementada del algoritmo DASG sólo puede aplicarse directamente a las funciones especificadas por su tabla de verdad cuando el número de variables de entrada es menor o igual a 21. Esta limitación es debida a que los recursos de CPU y memoria necesarias aumenta exponencialmente con el número entradas, ya que el algoritmo DASG trabaja directamente con la tabla de verdad de la función, tabla que contiene 2^N filas, siendo N el número de entradas. Sin embargo, una ventaja de este algoritmo es que puede ser paralelizado para obtener mejores resultados, ya que cada división es independiente de las demás y podrían realizarse como tarea independiente en diferentes procesadores.

Para conseguir que el algoritmo DASG pueda ser utilizado con funciones con un número mayor de entradas se realizó un preprocesamiento inicial de la función, descomponiéndola en un circuito de funciones booleanas con un *fan-in* máximo de 21 que implementa la función objetivo deseada. Este preprocesamiento de la función booleana se realizó con el programa SIS [**?**] utilizando el script *script.rugged* y los comandos *map -m 0 y reduce_depth* iterativamente. Estos comandos generan nuevos circuitos que implementan la misma función reduciendo el *fan-in* y el número de niveles. Los resultados se muestran en la Tab. 1.5, donde las características de las arquitecturas generadas se comparan con los obtenidos por Zhang. Se pueden observar resultados similares a los obtenidos previamente con funciones de hasta 21 entradas (ver Tab. 1.4), donde se obtuvo una reducción en el número de puertas y en el número de niveles (capas) a costa de un aumento de la interconexión. La reducción en el número de puertas fue del 10.35 %, ligeramente inferior al de la aplicación del algoritmo DASG directamente a la tabla de verdad de toda la función, donde se obtuvo una reducción del 20 %. En el número de niveles, las arquitecturas obtenidas

generan un media de 5.5 capas, reducción que es todavía considerablemente alta (40.7 %). El promedio de *fan-in* de las arquitecturas obtenidas fue del 7.9 con valores en el rango [4,15].

1.5. DASG*: Extensión del algoritmo DASG para funciones parcialmente definidas

Con el fin de analizar la capacidad de generalización del algoritmo DASG, se modificó el algoritmo añadiendo el símbolo $*$ al acrónimo para poder trabajar con los valores indeterminados. El uso del símbolo $*$ representa los casos desconocidos en el vector de la verdad de una función booleana (Tab. 1.6). El esquema general del procedimiento de descomposición, que se muestra en la Fig. 1.5, sigue siendo similar, aunque el uso de símbolos $*$ hace necesaria la introducción de modificaciones en el procedimiento de división de la función. Cuando una función (F^k) se descompone usando la variable *no-unate* con una influencia mas alta, la división genera dos funciones más simples, F_b^{k+1} y F_b^{k+1}, obtenidas de acuerdo a las dos siguientes ecuaciones:

$$F_a^{k+1}(\mathbf{X}) = \begin{cases} F^k(\mathbf{X}) & si\, x_i = 0 \\ t & si\, x_i = 1 \wedge F^k(\mathbf{X}) = t \\ * & en\, otro\, caso \end{cases} \tag{1.9}$$

$$F_b^{k+1}(\mathbf{X}) = \begin{cases} t & si\, x_i = 0 \wedge F^k(\mathbf{X}) = t \\ F^k(\mathbf{X}) & si\, x_i = 1 \\ * & en\, otro\, caso \end{cases} \tag{1.10}$$

donde

$$t = \begin{cases} 0 & si\, función\, de\, salida = OR \\ 1 & si\, función\, de\, salida = AND \end{cases}$$

De esta manera, si la función utilizada en la capa de salida es una $OR(AND)$ y la salida de la función para una determinada entrada $\mathbf{X}$ es $F(\mathbf{X}) = 0$ ($F(\mathbf{X}) = 1$), para que la salida de la red DASG sea $0\,(1)$, las dos funciones en las que se descompone la función $F_a(\mathbf{X})$ y $F_b(\mathbf{X})$ han de tener como salida $0\,(1)$. Ahora bien, si la salida de la función es $F(\mathbf{X}) = 1$ ($F(\mathbf{X}) = 0$), es necesario que sólo una de las funciones $F_a(\mathbf{X})$ y $F_b(\mathbf{X})$ tenga el valor a $1\,(0)$. De esta manera, las funciones por las que se divide la función $F^k(\mathbf{X})$ son menos restrictivas (una función parcial booleana con K símbolos $*$, representa a 2^K funciones booleanas diferentes), y sigue implementando la función original.

Las formulas introducidas en la ecuaciones 1.9 y 1.10 son también válidas para el problema de la síntesis de arquitecturas (sec. 1.3) analizado anteriormente.

1.5.1. Simplificación de una función

Una función booleana parcialmente definida puede ser simplificada eliminando una variable x_i si la influencia de la variable i-ésima es cero. Este caso ya no es tan trivial como en el sec. 1.3.2, ya que si la influencia de la variable i-ésima es cero, es debido a que no existe ningún efecto de la variable en la salida de la función. Esto puede ser debido a varios motivos; por un lado, al igual que el anterior caso, los cambios de la variable en la entrada no afectan en nada a la salida de la función, es decir, $f(,...,x_{i-1},0,x_{i+1},...) = f(...,x_{i-1},1,x_{i+1},...)$, aplicándose por lo tanto la ecuación para simplificar una función f de n entradas $f(...,x_{i-1},x_{i+1},...) = f(...,x_{i-1},0,x_{i+1},...)$; y por otro lado, es posible que la sensitividad sea cero por que la función no esté definida en alguna de las entradas.

La siguiente ecuación muestra como quedaría definida la simplificación de una variable de una función si su sensitividad es cero.

$$f(...,x_{i-1},x_{i+1},...) = \begin{cases} f(...,x_{i-1},0,x_{i+1},...) & si\,(f(...,x_{i-1},0,x_{i+1},...) = \\ & f(...,x_{i-1},1,x_{i+1},...)) \\ f(...,x_{i-1},0,x_{i+1},...) & si\,(f(...,x_{i-1},1,x_{i+1},...) = * \\ & \wedge f(...,x_{i-1},0,x_{i+1},...) \neq *) \\ f(...,x_{i-1},1,x_{i+1},...) & si\,(f(...,x_{i-1},0,x_{i+1},...) = * \\ & \wedge f(...,x_{i-1},1,x_{i+1},...) \neq *) \\ * & si\,(f(...,x_{i-1},0,x_{i+1},...) = * \\ & \wedge f(...,x_{i-1},1,x_{i+1},...) = *) \end{cases}$$

x_0	x_1	x_2	x_3	*Salida*
0	0	0	0	1
1	0	0	0	0
0	1	0	0	*
1	1	0	0	1
0	0	1	0	1
1	0	1	0	0
0	1	1	0	*
1	1	1	0	1
0	0	0	1	*
1	0	0	1	1
0	1	0	1	1
1	1	0	1	*
0	0	1	1	*
1	0	1	1	1
0	1	1	1	0
1	1	1	1	1

Tabla 1.6: Tabla de la verdad de la función booleana parcialmente definida con vector de la verdad {1, 0, *, 1, 1, 0, *, 1, *, 1, 1, *, *, 1, 0, 1}. Se puede observar que el vector de la verdad de la función coincide con la columna salida de la tabla de verdad. El uso del símbolo * es necesario para representar los casos desconocidos.

1.5.2. Ejemplo de generalización

En la Fig. 1.9 se muestra un ejemplo de la aplicación del algoritmo DASG* para el caso de una función booleana parcialmente definida de cuatro variables (Tab. 1.6). Nótese que el ejemplo que se ha elegido en este caso es la función mostrada en la Tab. 1.6, a la cual se le han eliminado algunas entradas. Esta función es *no-unate*, por lo tanto, no es umbral, teniendo que ser descompuesta en dos funciones más simples, Para ello se calcula la influencia de cada una de las variables *no-unate* de la función con el objeto de decidir por qué variable se va a dividir la función. En este caso sólo existe una variable *no-unate*, x_0, por lo que se procederá a dividirla en dos funciones más simples. Tras esto, se procede a eliminar en cada una de ellas aquellas

variables que tengan sensitividad nula, simplificando así la función. Se comprobó que cada una de ellas era linealmente separable, terminando así la ejecución del algoritmo ya que no quedan funciones por analizar en el conjunto *workset*. La Fig. 1.9 muestra la arquitectura final y la función final obtenida, incluyendo los ejemplos parciales no definidos al comienzo del problema. En la Tab. 1.7 se puede observar que de los 5 patrones que se deseaban clasificar inicialmente, sólo uno fue clasificado erróneamente, obteniéndose por lo tanto un porcentaje de acierto del 80 %.

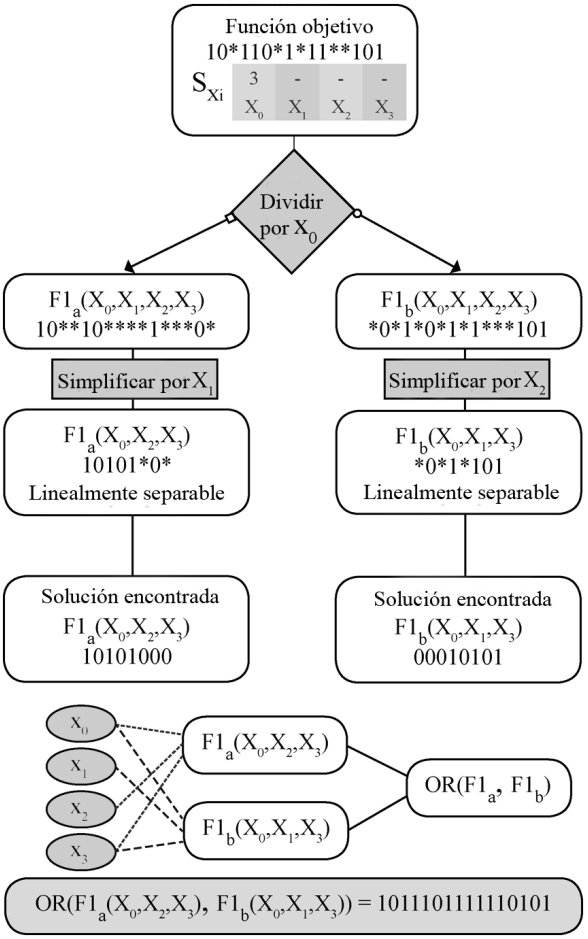

Figura 1.9: Ejemplo de la aplicación del algoritmo de generalización para el caso de una función parcialmente definida (Tab. 1.6).

Para probar la capacidad de generalización del algoritmo DASG*, se analizó un conjunto de 22 funciones con un número de entradas en el intervalo [5,14]. La elección

Entradas				Función	Función	Resultado obtenido
x_0	x_1	x_2	x_3	objetivo	deseada	con DASG*
0	0	0	0	1	1	1
1	0	0	0	0	0	0
0	1	0	0	*	0	1
1	1	0	0	1	1	1
0	0	1	0	1	1	1
1	0	1	0	0	0	0
0	1	1	0	*	1	1
1	1	1	0	1	1	1
0	0	0	1	*	1	1
1	0	0	1	1	1	1
0	1	0	1	1	1	1
1	1	0	1	*	1	1
0	0	1	1	*	0	0
1	0	1	1	1	1	1
0	1	1	1	0	0	0
1	1	1	1	1	1	1

Tabla 1.7: Resultados de clasificación obtenidos con el algoritmo DASG* para la función definida en la Tab. 1.6, a través del procedimiento mostrado en la Fig. 1.9

de este subconjunto de funciones de la Tab. 1.4 es debido a que con el resto de las funciones booleanas se obtuvo un 100 % de acierto en su generalización sobre el conjunto de test, incluso en el caso de utilizar solo un 20 % de los ejemplos como entrenamiento.

Existen muchos algoritmos de clasificación en la literatura, de los que hemos seleccionado tres alternativas de diferentes enfoques ampliamente utilizados: el algoritmo de arboles de decisión C4.5 [**?**], redes neuronales de tipo $feedforward$ (FFNN) entrenadas con el algoritmo de retropropagación (*backpropagation*), y una implementación del algoritmo de "vecino más cercano" para la generalización ($NN-gen$). Todos los métodos, excepto el algoritmo $DASG*$, se aplicaron usando la herramienta $WEKA$ [**?**] con los valores por defecto de los parámetros de cada uno de los métodos.

Función	Número entradas	DASG*	C4.5	FFNN	NN-Gen
cm82af	5	69.23	38.46	53.85	69.23
cm82ag	5	38.46	38.46	38.46	30.77
cm82ah	5	76.92	69.23	100.00	76.92
majority	5	61.54	69.23	61.54	61.54
z4ml24	7	96.08	76.47	96.08	82.35
z4ml25	7	96.08	70.59	100.00	39.22
z4ml26	7	100.00	56.86	100.00	86.27
z4ml27	7	100.00	60.78	100.00	100.00
9symml	9	91.71	82.93	100.00	79.51
alu2k	10	100.00	92.20	75.61	78.05
alu2l	10	81.71	82.20	68.78	73.66
alu2o	10	85.37	85.61	83.90	82.44
x2n*	10	98.05	98.29	98.53	98.29
x2p*	10	90.72	96.46	96.21	86.32
x2q*	10	96.58	84.98	97.56	90.23
cm85al	11	100.00	97.68	99.15	97.07
cm85am	11	99.51	97.31	98.29	96.58
cm85an	11	99.75	97.80	99.76	97.56
alu4q	14	99.57	98.32	86.15	95.76
alu4r	14	96.34	95.45	87.27	94.69
alu4u	14	97.68	97.76	95.44	96.67
cm162aq*	14	99.87	99.64	99.98	99.33
Media		89.78	81.21	88.03	82.38

Tabla 1.8: Capacidad de generalización obtenida por los algoritmos DASG*, C4.5, Perceptron multicapa, y KNN sobre un conjunto de 22 funciones booleanas.

El 60 % del número total de ejemplos se utilizaron como conjunto de entrenamiento, mientras que el 40 % restante se utilizó para probar la capacidad de generalización. En las 4 funciones marcadas con un asterisco, el conjunto de entrenamiento fue del 20 % (80 % para test) porque de lo contrario la capacidad de generalización alcanzaba el 100 % para todos los métodos utilizados.

En la Tab. 1.8, se muestran los resultados obtenidos mediante los cuatro métodos. Los mejores resultados fueron obtenidos por el algoritmo $DASG*$ con un promedio generalización de 89.78 %, seguido de cerca por FFNN (88.03 %), C4.5 (81.21 %),

y KNN (82.38 %), resultando estadísticamente significativo las diferencias entre DASG* y los algoritmos C4.5 y KNN (*p=0.009* y *p=0.012* respectivamente).

1.6. Conclusión

En este capítulo se ha introducido el algoritmo de redes neuronales constructivo DASG (*Descomposition Algorithm to Synthesis and Generalization*)[**?**]. DASG es un algoritmo que genera arquitecturas de red compactas compuestas por una única capa oculta constituida por puertas umbrales y una neurona en la capa de salida que implementa la función *OR* o la función *AND* según necesidad. En esencia, el algoritmo reduce la complejidad de una determinada función booleana, denominada función objetivo (F_{obj}), descomponiéndola en otras dos funciones más simples, F_a y F_b, según la variable de mayor influencia. Este proceso se repite recursivamente hasta encontrar un conjunto de funciones linealmente separables tales que su unión con el operador *OR* o con el operador *AND* genere nuevamente la función objetivo. En la práctica, los costes computacionales restringen el número de entradas de la red, ya que la tabla de la verdad sobre la que trabaja el algoritmo crece exponencialmente con el número de entradas. Sin embargo, el algoritmo puede ser aplicado a funciones con un número muy grande de entradas descomponiéndolas previamente en funciones más pequeñas.

Para probar el rendimiento del algoritmo, se aplicó inicialmente a un subconjunto de funciones de referencia de hasta 21 entradas, y luego se analizó el rendimiento en funciones de hasta 257 entradas mediante la aplicación del algoritmo en combinación con un preprocesamiento utilizando el sistema SIS. Los resultados se compararon con los obtenidos por Zhang et al. [**?**] (ver Tab. 1.4 y Tab. 1.5), y muestran que el nuevo algoritmo es muy eficiente en términos del número de puertas (reducción del

19.91 %, 85 % en el mejor caso) y niveles de las arquitecturas construidas (reducción del 61.87 %). Sin embargo, hay que señalar que las arquitecturas construidas con el algoritmo *DASG* tienen en promedio un mayor número de conexiones. Cuando el algoritmo *DASG* se aplicó en combinación con SIS a funciones de mayor tamaño, las mejoras en el número de niveles y el número de puertas de las arquitecturas eran menores, pero en cualquier caso significativas, con un promedio de 10.35 % de reducción en el número de puertas y un 40.7 % en el número de niveles. Para este segundo conjunto de funciones, el nivel de interconexión también fue mayor que los valores reportados por Zhang et al. [**?**] con un incremento de hasta el 15.67 % .

Un aspecto importante de las arquitecturas generadas por el algoritmo *DASG* refiere a los valores máximos de *fan-in* para los circuitos obtenidos, con un valor promedio de 13 para el primer conjunto de funciones (entradas hasta 21), y un valor de 7.9 para el segundo conjunto de funciones (hasta 257 entradas). El método implementado por Zhang [**?**] no mejora si el *fan-in* permitido máximo se incrementa más allá de 6.

Con el fin de analizar la capacidad de generalización del algoritmo DASG, el algoritmo ha sido modificado añadiendo el símbolo $*$ para poder trabajar con valores indeterminados (algoritmo DASG*). En la Tab. 1.8 se muestran los resultados de la aplicación de DASG* a un conjunto de funciones pertenecientes al conjunto de datos de referencia *MCNC*, y se compararon con los resultados obtenidos mediante la aplicación de tres diferentes algoritmos de clasificación: C4.5 árboles de decisión, redes neuronales $feedforward$ estándar (FFNN), y clasificadores de vecinos más cercanos (KNN). La capacidad de generalización del algoritmo DASG* es la mayor entre los cuatro métodos analizados, logrando un valor promedio de aciertos de 89.78 %. FFNN obtuvo una capacidad de generalización ligeramente inferior, aunque no estadísticamente significativa (88.03 %), mientras que para los otros dos métodos, árboles de decisión C4.5 y KNN, la generalización obtenida fue menor y

estadísticamente significativa (81.21 % y 82.38 %, respectivamente).