Universidad Zaragoza
1542

Proyecto Fin de Carrera

UNA APROXIMACIÓN A LA OPTIMIZACIÓN DE ALGORITMOS MEDIANTE EL USO DE MINIMIZACIÓN DE FUNCIONES BOOLEANAS

Autor

Rubén Alejandro Escartín Aparicio

Directores

Javier Campos Laclaustra
Víctor Viñals Yúfera

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza
1542

ESCUELA DE INGENIERÍA Y ARQUITECTURA DE ZARAGOZA

2017

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Agradecimientos

Gracias a mis profesores.

A Javier Campos por ser el mejor profesor que he tenido durante la carrera y permitirme darle salida a esta locura.

A Víctor Viñals por guiarme con inteligencia.

A todos los profesores que he tenido en la universidad, de todos he aprendido algo.

Mención especial a Pedro Muro por introducirme a Lisp.

Gracias a mi familia.

A mi padre por su esfuerzo, sin el que nunca podría haber estudiado una ingeniería.

A mi hermana por todas las ocasiones en las que ha estado a mi lado.

A Pilar y su familia, porque estos años se ha convertido en la mía.

Gracias a mis amigos, sabéis quienes sois: Jorge, Lucía, Irene, Héctor…

Gracias a todos los compañeros que he conocido durante estos años.

Esto, en cierta forma, es de todos.

Y gracias a mi cabezonería, por no permitirme renunciar nunca.

Una aproximación a la optimización de algoritmos mediante el uso de minimización de funciones booleanas

RESUMEN

Este proyecto explora la posibilidad de optimizar algoritmos utilizando técnicas de minimización de funciones booleanas.

La idea de partida es que expresar un programa a muy bajo nivel permitirá localizar y eliminar redundancia. Para ello se trabaja con operaciones bit a bit de lógica booleana. Usamos únicamente la función NAND para expresar cualquier otra función gracias a su propiedad de completitud funcional.

Expresar un algoritmo de esta forma nos permite, por un lado, tener una medida del coste del algoritmo en funciones NAND y, por otro, paralelizarlo.

Mediante minimización, se puede optimizar un circuito lógico equivalente a un fragmento de código secuencial, que no tenga bucles ni recursividad. Para ello se ha desarrollado una técnica propia de minimización rápida.

Se han desarrollado técnicas para este proyecto que permiten aplicar la minimización a algoritmos recursivos. De este modo se eliminan, por ejemplo, operaciones repetidas en diferentes iteraciones de un bucle.

Para llevar a cabo este trabajo se ha desarrollado una notación propia, parecida a un lenguaje ensamblador, que permite trabajar con funciones lógicas y recursividad.

Se ha creado una base de datos dónde se definen las funciones recursivas, que pueden representar desde una puerta lógica hasta un algoritmo como el de la suma.

Se han implementado los métodos de optimización de estas funciones recursivas y un método de evaluación, mediante el que se ejecutan para comprobar que son correctas. También se han implementado una serie de utilidades para, por ejemplo, traducir entre diferentes notaciones.

Finalmente se han comparado los resultados con el algoritmo sin optimizar y con la solución que nos ofrecerían otras herramientas.

Índice

"La escultura ya estaba dentro de la piedra. Yo, únicamente, he debido eliminar el mármol que le sobraba"- Anónimo

1 Introducción

1.1 Motivación y contexto

Este proyecto es un proyecto de iniciación a la investigación. Nace a propuesta del alumno, a partir de la intuición de que frente al problema de optimización combinatoria que representa elegir las instrucciones más adecuadas para implementar un algoritmo, una posible aproximación es expresarlas todas a más bajo nivel, en un lenguaje de una sola instrucción.

La motivación fundamental del proyecto es la observación de que existe un problema de difícil solución: los algoritmos rara vez son óptimos. La optimización automática de código se deja actualmente en manos de los compiladores. Pero ningún compilador puede garantizar que se obtiene el código más rápido o más pequeño, ya que para hacerlo tendría que resolver el problema de parada [1], que es indecidible.

En los compiladores tradicionales se optimiza mediante la aplicación de una serie de reglas fijas, en un orden determinado. Estas reglas y el orden en que se aplican suelen ser lo suficientemente buenas para producir un código de alto rendimiento, pero eso no quiere decir que no se pueda optimizar más.

1.2 Antecedentes y estado de la técnica

La optimización tiene muchas vertientes. Hay problemas de optimización de código que son NP-completos o incluso indecidibles. Este proyecto se centra única y exclusivamente en la optimización del procesamiento aritmético-lógico.

Algunas técnicas habituales de los compiladores son el "inlining" de funciones, la eliminación de código muerto, análisis de flujo, selección de instrucciones máquina complejas, etc. Otra de estas reglas es la supresión de subexpresiones comunes, donde se eliminan operaciones duplicadas [2]. Algunas de estas optimizaciones se aplican una vez el compilador ha generado su código intermedio y otras en fases posteriores, pero siempre se hace sobre un programa expresado en repertorio complejo de sentencias o instrucciones.

No se han encontrado en la literatura técnicas de minimización de circuitos aplicadas para optimizar código.

1.3 Objetivo y alcance

El objetivo del proyecto es explorar la posibilidad de optimizar código con un enfoque diferente. A fin de validarlo se ha desarrollado un software optimizador.

Para ello se expresan algoritmos al nivel más bajo posible y se elimina la redundancia a ese nivel.

El método de optimización empleado implica transformar el código objeto a un código intermedio y optimizar este código intermedio. Se utilizará un lenguaje intermedio basado únicamente en la función lógica booleana NAND y la recursividad. Definimos optimizar como reducir una operación al mínimo número de operaciones lógicas NAND equivalentes.

Para optimizar este código intermedio, se sintetizan circuitos de lógica booleana a partir del algoritmo inicial y se minimizarán estos circuitos.

Como paso adicional se podría volver a traducir este código intermedio al código objeto inicial.

1.4 Métodos y técnicas

Un fragmento de código secuencial es equivalente a un circuito lógico. A ese circuito se le puede aplicar una técnica de minimización de funciones booleanas. Con eso se obtiene un circuito simplificado, que es equivalente al fragmento de código secuencial original optimizado a muy bajo nivel. El proceso es similar a la supresión de subexpresiones comunes, eliminando operaciones lógicas bit a bit redundantes.

Este proyecto pretende aprovechar la transformación del código a tan bajo nivel para optimizar y paralelizar algoritmos en la medida de lo posible.

En 2008 se demostró que el problema de la minimización de circuitos de funciones booleanas es $\sum_2^P$ −completo [3]. La aproximación para resolver este problema ha de ser por lo tanto heurística.

La implementación del método propuesto para optimizar algoritmos consta de una **base de datos** donde se almacenan definiciones. Cada definición se define recursivamente mediante otras definiciones. Todas las definiciones se construyen a partir de una definición base que es la operación lógica NAND. La operación NAND representa una puerta lógica universal, mediante la cual es posible realizar todas las demás operaciones lógicas. Estas definiciones se pueden usar para expresar tanto algoritmos como instrucciones de código objeto.

Para optimizar un algoritmo, creamos una definición que lo representa, con sus entradas y salidas. Al construir todas las definiciones a partir de la definición NAND, pueden expresarse únicamente mediante esta función y la recursividad. Este es el proceso de **traducción** a un **lenguaje intermedio**.

Para optimizar un algoritmo secuencial, se traduce a un **bosque[1] de instrucciones lógicas NAND**, en un proceso similar a la síntesis de un circuito lógico. La optimización de este bosque es equivalente a minimizar este circuito lógico, que en el caso de muy pocas variables sabríamos resolver mediante un mapa de Karnaugh. Tradicionalmente, para la minimización de circuitos lógicos se aplican algoritmos heurísticos como Espresso [4]; en su lugar, aplicaremos una técnica heurística más sencilla, de coste lineal. La técnica consiste en eliminar las operaciones duplicadas y la doble negación del circuito. Esta reducción es el núcleo del proceso de optimización.

Para optimizar un algoritmo recursivo se optimiza, tanto la parte secuencial de este, como las operaciones redundantes entre iteraciones recursivas.

En cuanto al apartado técnico, el optimizador está escrito en Java, que es un lenguaje portable, muy usado y con excelentes herramientas para la depuración. Se han desarrollado algoritmos para traducir código objeto a código intermedio, así como para optimizarlo.

Se ha profundizado en el conocimiento adquirido en las asignaturas de Compiladores respecto a los lenguajes intermedios y en los conocimientos adquiridos en las asignaturas del área de Arquitectura de Computadores en cuanto a circuitos y su minimización. Además, hemos usado estructuras de datos arborescentes y múltiples tablas dispersas, aplicando los conocimientos adquiridos en Estructuras de Datos y Algoritmos.

1.5 Estructura del resto de la memoria

En primer lugar, plantearemos de la forma más sencilla posible el problema al que nos enfrentamos y los requisitos necesarios para abordarlo. En segundo lugar, explicaremos la solución adoptada, empezando por la notación empleada, con las estructuras de datos elegidas y los métodos utilizados.

[1] Un bosque es un conjunto de árboles. [17]

Finalmente, compararemos los resultados obtenidos con los proporcionados por una herramienta de minimización de circuitos y extraeremos conclusiones.

2 Planteamiento del problema

"La belleza es la purgación de lo superfluo."

- Miguel Ángel

Para abordar el problema de optimizar algoritmos, lo primero que necesitaremos es un modelo para representarlos.

Representaremos los algoritmos mediante funciones recursivas, por lo que a partir de ahora se usarán ambos términos indistintamente.

El modelo de representación elegido está formado por la función lógica universal NAND y la definición de nuevas funciones. Las nuevas funciones se podrán definir a partir de las funciones que ya estén definidas y podrán ser recursivas.

Además, será necesario representar de alguna forma los datos de entrada y salida. Para ello utilizaremos *nodos*.

Como estamos trabajando con funciones recursivas, necesitaremos que esos nodos puedan dividirse en *subnodos*.

Dado que usamos definiciones de funciones a partir de otras funciones, será necesaria una base de datos donde almacenarlas.

Para resolver este problema necesitaremos representar el algoritmo original, identificar los segmentos secuenciales del algoritmo, sintetizar el circuito que los represente, minimizarlo y aplicar las optimizaciones obtenidas al algoritmo.

Además, buscaremos otras formas de aplicar esta optimización de código secuencial a funciones recursivas.

3 Solución del problema

"Everything should be made as simple as possible, but not simpler." – attributed to Albert Einstein

3.1 Notación utilizada

La notación tradicional usada en lógica puede resultar bastante redundante, por ejemplo al intentar representar la función XOR mediante funciones NAND:

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

Para representar una expresión arbitraria, con un número ilimitado de entradas y salidas locales, usaremos la siguiente notación:

FUNCIÓN [NodoEn1, NodoEn2, … ; NodoS1, NodoS2,…] =

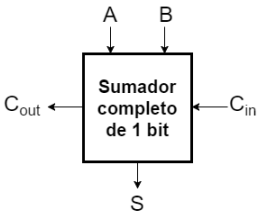
FUNCIÓN1 [NodoEnL1, NodoEnL2, … ; NodoSL1, NodoSL2,…],

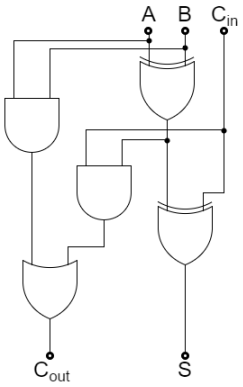
FUNCIÓN2 [NodoEnL1, NodoEnL2, … ; NodoSL1, NodoSL2,…]…

Donde:

- Los nodos representan literales. Un literal es una variable o su complemento.
- NodoEn1, NodoEn2, … representan nodos de entrada de la función principal
- NodoS1, NodoS2 representan nodos de salida de la función principal
- FUNCIÓN1, FUNCIÓN2, …. representan instancias de funciones usadas para definir la función principal
- NodoEnL1, NodoEnL2, … representan nodos de entrada locales
- NodoSL1, NodoSL2, … representan nodos de salida locales

Dado un sumador completo de 1 bit:





A	B	C_{in}	S	C_{out}
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	1	1	1

Figura 1: Sumador completo de 1 bit

$$C_{out} = (A \oplus B)C_{in} + AB$$
$$S = A \oplus B \oplus C_{in}$$

Podemos representarlo como:

Sum[A,B,C_{in}; S,C_{out}] =

XOR[A,B; t1], AND[t1,C_{in}; t3], AND[A,B; t2], OR[t2,t3; C_{out}], XOR[A,B; t4], XOR[t4, C_{in}; S]

Además, es posible separar en líneas las funciones según las dependencias que se producen entre sus nodos. Toda función local, que usa un nodo de entrada, deberá ir siempre después de cualquier otra que tenga ese mismo nodo como salida.

Sum[A,B,C_{in}; S,C_{out}] =

XOR[A,B; t1], AND[A,B; t2], XOR[A,B; t4]

AND[t1, C_{in}; t3], XOR[t4,C; S]

OR[t2,t3; C_{out}]

De esta forma, estamos exponiendo el paralelismo de la definición de la función, ya que las operaciones que están en una misma línea podrían ejecutarse en paralelo.

Dado que las instancias pueden ser recursivas, encontramos la necesidad de dividir los nodos en subnodos.

La forma de indexar los subnodos de un nodo está inspirada en el lenguaje Lisp. Podemos hacer referencia al subnodo "final" o al "resto" de subnodos.

En Lisp la estructura de datos primaria es la lista. Se utilizan dos operaciones, "car" y "cdr" para seleccionar el primer elemento de la lista o el resto de la lista.

Nosotros hacemos algo parecido con los nodos: mediante "último" seleccionamos el último subnodo que pueda componer un nodo y mediante "resto" seleccionamos todos los demás.

En el caso de que el nodo no tenga subnodos definidos, estaremos seleccionando los bits ("final" o "resto") que componen ese dato.

Por lo tanto:

- Para indicar el subnodo final de un nodo, usaremos X{n}
- Para indicar el resto de subnodos, usaremos X{1..n-1}

Por ejemplo, si queremos aplicar la función lógica AND entre un bit y un vector de bits, podemos imaginar el siguiente circuito:

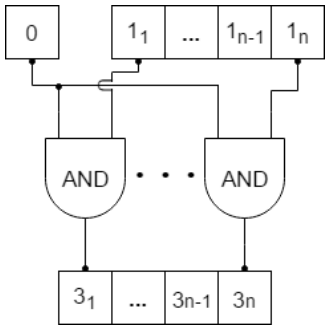


Figura 2: Circuito de funciones AND entre un bit y un vector de bits

Para representarlo, definiremos la función recursiva:

andRecursivo[0,1; 3(5&4)] =

and [0,1{n}; 4]

andRecursivo[0,1{1..n-1}; 5]

Se utiliza la notación de índice matemático (empezando en el 1) en lugar del convenio usado en los índices en informática (empezando en el 0) para facilitar su lectura.

Una desventaja de esta forma de representar los datos es que acceder al primer nodo de una lista tendría coste lineal, ya que no tenemos acceso directo y habría que iterar todos los nodos uno a uno. Sin embargo, esto es sólo un modelo para representar un algoritmo, de modo que el coste real no tiene por qué ser ese.

3.2 Estructuras de datos empleadas

3.2.1 Representación de valores binarios

Utilizamos un tipo de datos para almacenar cadenas de bits, que representan datos procesados por las funciones. Java SE dispone de la clase BitSet; el problema es que la longitud de las cadenas depende siempre del número de bits significativos que contenga. Por la naturaleza de las operaciones que realizamos, necesitamos controlar el número de bits que tiene una cadena y que no tiene porqué ser el mismo que el de bits significativos. Por ejemplo, una representación del valor "0" en una arquitectura de 8 bits ha de ser "00000000". Para ello se ha ampliado la clase BitSet de Java SE, añadiendo un campo de longitud.

A esta clase ampliada la llamamos fixedBitSet.

También se ha implementado la función NAND que, dados dos objetos fixedBitSet, nos devuelve un tercer objeto, resultado de aplicar esta operación.

3.2.2 Base de datos de funciones

A fin de trabajar con funciones con la notación descrita, se ha implementado una base de datos de funciones.

La función NAND está predefinida y se utiliza para definir cualquier otra función. Cada nueva función que se añade a la base de datos se define mediante otras que ya estén en la base de datos.

No se permite usar funciones que no estén definidas antes en la base de datos, salvo la función que se está definiendo. De esta forma se permite la recursividad, pero se elimina la posibilidad de que se dé recursión indirecta.

Sí, por ejemplo, añadimos la definición de la función NOT a la base de datos, nos quedará:

BASE DE DATOS:

not[0; 1] =

nand [0,0; 1]

nand[0,1; 2] =

nand [0,1; 2]

3.2.2.1 *Funciones*

Las funciones están compuestas por nodos e instancias de otras funciones y su nombre tiene que ser único. Los nodos representan los datos. Cada función tiene una serie de nodos de entrada y una serie de nodos de salida.

Las instancias se almacenan en una estructura bidimensional, para poder ordenarlas según las dependencias entre sus nodos. Si un nodo es salida de una instancia, todas las instancias a funciones que lo utilicen como entrada irán después.

Cada función almacena cuáles son las instancias recursivas (de sí misma) y qué instancias contienen recursividad. Esto es así para facilitar el tratamiento posterior de funciones recursivas.

Un ejemplo de función es:

or[0,1; 2] =

not [0; 3] not [1; 4]

nand [3,4; 2]

3.2.2.1.1 Instancias

Las instancias son referencias a funciones que se usan en la definición de la función principal.

En el ejemplo anterior NOT y NAND son instancias a funciones usadas para definir la función principal OR.

Una instancia tiene que tener, por lo tanto, una serie de nodos de entrada, una serie de nodos de salida y una función, que es la función instanciada. Además, almacenamos la profundidad a la que se sitúa esa instancia dentro de la estructura de datos bidimensional de instancias de una función, para facilitar la inserción de nuevas instancias.

3.2.2.1.2 Nodos

Los nodos representan los datos empleados en las funciones.

Un nodo debería tener un subnodo “final” y un subnodo “resto”.

Cada nodo almacena también su nodo padre, si lo hay, si es salida de una instancia y a qué definición principal pertenece.

3.2.2.1.2.1 *Problemática de los nodos*

En ocasiones necesitaremos que un nodo, hijo de algún nodo ya definido, componga un nuevo nodo.

Este caso se da por ejemplo en la definición usada para la suma, que veremos en la Sección 4.3: tenemos el nodo 10(8{n}&7&11) que necesita para su definición del nodo 8{n}.

3.2.2.1.2.2 *Solución propuesta a la problemática de los nodos*

Hasta ahora usábamos los subnodos “final” y “resto” para designar elementos de un nodo.

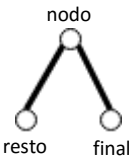


Figura 3: Nodo con subnodos "resto" y "final".

La solución adoptada ha sido utilizar los subnodos “hijoFinal” y “restoDeHijos” para elegir elementos y por otro lado unos subnodos “padreFinal” y “restoDePadres” para almacenar la lista de elementos.

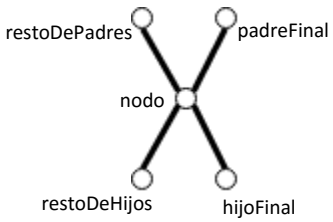


Figura 4: Nodo con hijos "restoDeHijos" e "hijoFinal" y padres "restoDePadres" y "padreFinal".

De esta forma los nodos padres cumplen una función parecida a la yuxtaposición usada en la notación ("&") y los nodos hijos a los subíndices ({1..n-1} y {n}). Cómo en Lisp, encadenando nodos podremos representar listas y seleccionar cualquier elemento de ellas.

3.2.3 Bosque de funciones NAND

Además de las estructuras usadas para representar funciones, usamos una estructura de datos adicional durante el proceso de minimización.

La estructura es un bosque, es decir, un conjunto de árboles de funciones NAND. En este caso los árboles pueden tener nodos en común.

Un bosque de funciones NAND tiene, por lo tanto, una serie de nodos de entrada y una serie de nodos de salida. Se utiliza, además, un contador de nodos para facilitar las tareas de representación.

3.2.3.1 *Nodos NAND*

Todos los nodos de esta estructura son indivisibles y, por ello, de igual tamaño.

Cada nodo de esta estructura es, o bien una entrada del bosque de funciones NAND, o bien la salida de una función NAND.

Por lo tanto, cada nodo tiene referencias a dos nodos padres, formando así una función NAND de dos entradas y una salida. Estas referencias están vacías para los nodos de entrada.

El bosque que representa la función NOT es, por ejemplo:

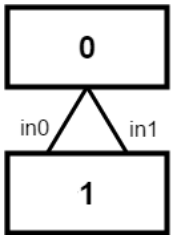


Figura 5: Representación de la función NOT mediante un bosque de funciones NAND

Donde el nodo 0 es el único nodo de entrada, el nodo 1 es el único nodo de salida y las dos entradas del nodo 1 corresponden al nodo 0.

3.3 Técnica propia de minimización de circuitos

La aproximación moderna a la minimización de circuitos la separa en dos problemas: optimización independiente de la tecnología y optimización dependiente de la tecnología. [5] El método aquí propuesto se centra únicamente en la optimización independiente de la tecnología.

Dentro de la optimización independiente de la tecnología existen dos aproximaciones: la minimización en dos niveles y la minimización multinivel.

La minimización en dos niveles busca una expresión mínima en suma de productos (OR de ANDs) o producto de sumas (AND de ORs). El objetivo es reducir el número de literales y productos (o sumas) utilizados.

La minimización multinivel minimiza una expresión con funciones lógicas arbitrarias. El objetivo es reducir únicamente el número de literales utilizados.

La minimización en dos niveles tiene un problema de compromiso: minimiza el retraso del circuito a costa de maximizar el área (puertas y literales).

La minimización multinivel aparece para solucionar ese compromiso y permitir un mayor retraso minimizando el área del circuito.

El ejemplo clásico de minimización en dos niveles es el uso de un mapa de Karnaugh para expresar la función algebraica booleana representada como el sumatorio de sus minitérminos o producto de sus maxitérminos. Este método aprovecha las capacidades humanas de detección de patrones.

El método de Quine–McCluskey es un método determinista en dos niveles para encontrar la minimización de funciones booleanas como sumas de productos. [6] Este método es más adecuado para implementarlo en computadores, pero es NP-complejo. El tiempo de resolución del algoritmo crece de forma exponencial con el aumento del número de variables.

El algoritmo Espresso es el estándar de la industria para la minimización de dos niveles. [5] Se trata de un algoritmo heurístico, que no tiene por qué encontrar la solución óptima, pero se comporta suficientemente bien. Fue desarrollado por Robert K. Brayton en la Universidad de California, Berkeley. [7]

El algoritmo Espresso-exacto (o “mincov”) es una implementación moderna del algoritmo de Quine–McCluskey, que mantiene el problema de crecimiento exponencial con el número de variables. [8] Mediante este algoritmo se puede hallar la solución exacta, pero solo es práctico con un número muy reducido de variables.

MisII es la segunda versión del algoritmo de minimización multinivel MIS, desarrollado también por Robert K. Brayton y utilizado por compañías como Intel o Dec [9]. MisII minimiza de forma dependiente de la tecnología y utiliza Espresso como una de sus subrutinas. Tanto Espresso como MisII pertenecen a la colección de programas y librerías OctTools de la Universidad de California, Berkeley, para el diseño de circuitos integrados. [10]

El método de *Transducción* es un método multinivel que minimiza los términos que no importan en redes compuestas únicamente de funciones NOR. [11] Tiene similitudes al método que aquí se presenta, como el hecho de aprovechar la universalidad de la puerta lógica NOR, y aplicar unas reglas de simplificación para eliminar redundancia; pero estas son mucho más complejas y el método es iterativo, mientras que nuestro método es de una sola pasada [4].

Todos los algoritmos de minimización mencionados utilizan una Red Booleana como medio para representar circuitos.

Nuestro método utiliza una representación alternativa de una Red Booleana, que describiremos mediante una notación propia. Se usan únicamente dos reglas para minimizar. El algoritmo tiene un coste medio de ejecución lineal en el número de literales. Para los ejemplos estudiados, este método minimiza igual o mejor que MisII, en el caso concreto de la representación de un circuito mediante puertas NAND de dos entradas.

3.3.1 Representación de la Red Booleana

Una Red Booleana es el modelo estándar independiente de la tecnología para representar circuitos mediante una red lógica. Los nodos de la red pueden ser entradas primarias, salidas primarias o funciones booleanas. Las funciones representan una expresión arbitraria, con un número ilimitado de entradas locales y una única salida [5] [12].

Un conjunto de operaciones lógicas se dice funcionalmente completo si cualquier función booleana puede ser expresada en términos de este conjunto de operaciones. Es habitual usar un conjunto de operaciones lógicas funcionalmente completo que incluya las operaciones AND, NOT, OR y XOR. Sin embargo, el conjunto unitario formado únicamente por la función NAND es funcionalmente completo, es decir, podemos representar cualquier función booleana arbitraria mediante funciones NAND. [13]

Se propone, por lo tanto, una Red Booleana, pero usando únicamente funciones NAND de dos entradas, que pueden representar expresiones arbitrarias. Este modelo presenta la ventaja de facilitar ciertas simplificaciones. Al utilizar una única función, se pretende evitar el problema de optimización combinatoria derivado del uso de varias funciones.

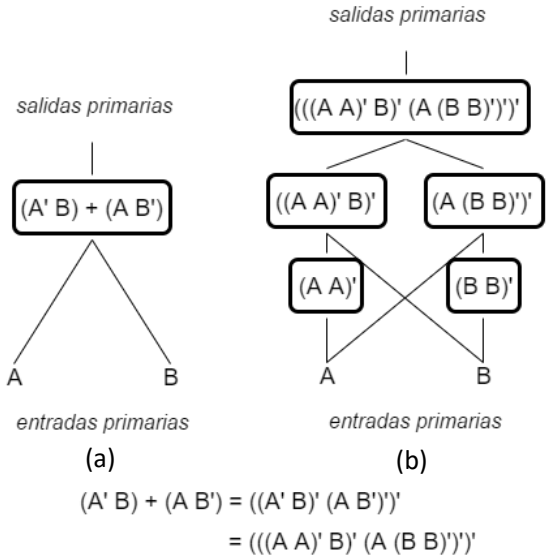


Figura 6 : Representación de la función XOR mediante Redes Booleanas

En la Figura 6 : Representación de la función XOR mediante Redes Booleanas(a) tenemos la representación de la función XOR mediante una Red Booleana de expresiones arbitrarias.

En la Figura 6(b) tenemos la representación de la función XOR mediante una Red Booleana compuesta únicamente de funciones NAND de dos entradas. Representaremos esta estructura mediante un bosque de funciones NAND, donde los nodos son los literales.

Este modelo podría considerarse dependiente de la tecnología, al limitar la representación de la red booleana a un tipo de puertas lógicas; sin embargo el planteamiento es el inverso: utilizamos las funciones NAND de dos entradas como el bloque que construye cualquier otra expresión.

Esto nos permite introducir una nueva métrica: el número de funciones NAND utilizadas en esta red (que es igual al número de literales que aparecen en las salidas de funciones) representa el coste de cualquier expresión arbitraria. Como ejemplo, podríamos decir que el coste en funciones NAND de dos entradas de la función XOR representada mediante el circuito de la Figura 3 es de 5 y su profundidad (que representa el retraso del circuito) 3.

El objetivo sigue siendo reducir el número de literales, aunque hay que tener presente que no todos los literales de una Red Booleana formada exclusivamente por funciones NAND serán literales de una Red Booleana de expresiones arbitrarias equivalente.

Una ventaja del modelo es que al haber una única operación, no hace falta representar diferentes operaciones; trabajamos únicamente con variables (los literales). Otra ventaja de este modelo es

que múltiples representaciones mediante expresiones arbitrarias de una Red Booleana darán lugar a la misma representación de una Red Booleana de funciones NAND.

3.3.2 Funcionamiento del algoritmo

Podemos encontrar cierto paralelismo entre el funcionamiento de nuestro algoritmo y Espresso.

Espresso es un algoritmo iterativo que repite una serie de fases: expansión, eliminación de la redundancia y reducción [5].

El algoritmo aquí presentado se centra en la eliminación de la redundancia. Comparándolo con Espresso, la expansión correspondería con la expresión de otras funciones como funciones NAND, y la reducción sería el proceso de volver a expresar las funciones NAND como funciones más complejas. Sin embargo, estas trasformaciones quedan fuera del modelo independiente de la tecnología.

El desarrollo de esta técnica se debe a la observación de que aplicar las dos leyes de De Morgan a una estructura formada únicamente por funciones booleanas NAND es muy sencillo.

El método de minimización es constructivo, es decir, partiendo de un circuito no minimizado, se va construyendo el circuito minimizado, agregando las funciones NAND necesarias una a una. Para obtener las funciones ordenadas desde las entradas hasta las salidas, se invocan sucesivamente los árboles que constituyen el bosque de funciones NAND que representan la Red Booleana, desde las hojas hacia las raíces.

3.3.2.1 *Primera regla: simplificación mediante la eliminación de la doble negación*

La primera regla de minimización consiste en eliminar la doble negación de funciones booleanas NAND.

3.3.2.1.1 Primera ley De Morgan

La primera ley De Morgan dice que $\overline{\overline{A} \cdot \overline{B}} = \bar{A} + \bar{B}$.

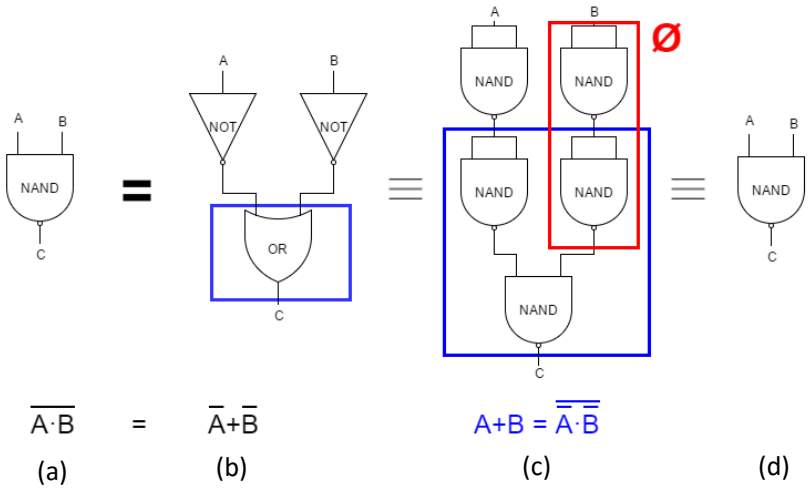


Figura 7: Aplicación de la primera ley de De Morgan mediante una red de funciones NAND

Los circuitos de las figuras Figura 7 (a) y Figura 7 (b) representan la equivalencia de la primera ley de De Morgan.

La Figura 7 (c) es el circuito resultante de la expresión en funciones NAND de la Figura 7 (b). Aplicamos la segunda ley De Morgan para expresar la función OR en funciones NAND.

$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

$$A + B = \overline{\bar{A} \cdot \bar{B}}$$

Por último, mediante la eliminación de la doble negación obtenemos la función original, Figura 7 (d).

Con esto queda demostrado que este método aplica la primera Ley de Morgan, haciendo uso únicamente de la eliminación de la doble negación.

3.3.2.1.2 Segunda ley De Morgan

La segunda ley de De Morgan es $\overline{A + B} = \bar{A} \cdot \bar{B}$

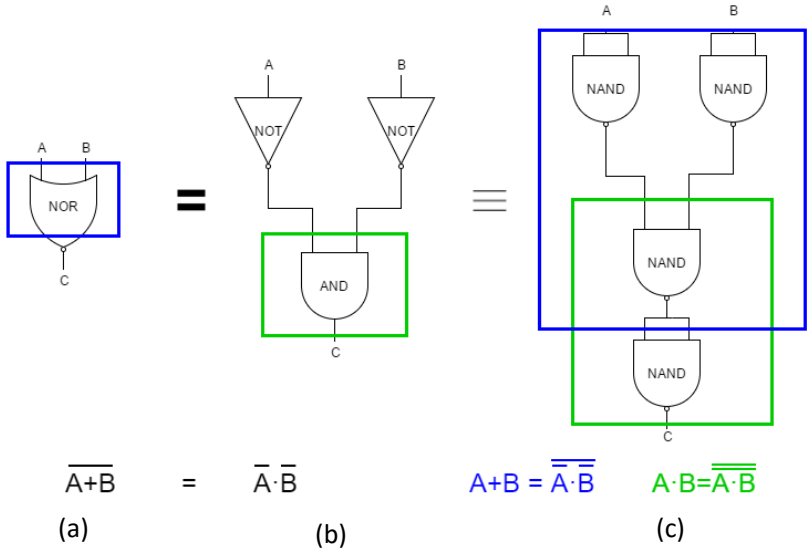


Figura 8: Aplicación de la segunda ley de De Morgan mediante una red de funciones NAND

Los circuitos de las figuras Figura 8 (a) y Figura 8 (b) representan la equivalencia de la segunda ley de De Morgan.

Aplicando la segunda ley De Morgan para expresar la función OR en funciones NAND, obtenemos el circuito de la figura Figura 8 (c) a partir del circuito de la figura Figura 8 (a).

Si expresamos en funciones booleanas NAND el circuito de la segunda ley de De Morgan, figura Figura 8 (b), obtenemos el mismo circuito de la figura Figura 8 (c).

Con esto queda demostrada la aplicación de la segunda ley de De Morgan mediante este método de minimización.

Nuestro método aplica, por lo tanto, al menos las dos leyes de De Morgan y la eliminación de dobles negaciones.

3.3.2.2 *Segunda regla: eliminación de funciones redundantes*

La segunda regla consiste simplemente en impedir que se defina una función NAND equivalente a alguna ya existente, es decir, con las mismas entradas.

Esto se puede implementar, de forma sencilla, mediante una tabla dispersa bidimensional cuyos índices sean los dos nodos de entrada. Además, estos dos nodos de entrada estarán ordenados para que no haya repeticiones de la misma función con las dos mismas entradas en orden inverso.

Por ejemplo, para la función AND, sabemos que:

$$ab = (ab)'' = ((ab)'(ab)')'$$

Luego en nuestra notación:

AND[a,b; c]=

NAND[a,b; x1], NAND[a,b; x2]

NAND[x1,x2; c]

Al minimizar esta definición, usamos una tabla dispersa bidimensional para asegurarnos de que no haya funciones repetidas. En el caso de que ya haya una función definida con esos dos nodos de entrada, la tabla devolverá como nodo de salida el que ya ha sido establecido. Además, se almacenará en otra tabla que el nodo eliminado ha de ser reemplazado por el nodo ya establecido; de esta forma se podrán corregir las siguientes referencias a este nodo.

Mediante esta tabla bidimensional se obtiene una definición libre de funciones NAND equivalentes redundantes.

NodoEnL1	NodoEnL2	NodoSL
a	b	x1
x1	x1	c

Nodo	Nodo equivalente
x2	x1

Figura 9: Tabla dispersa bidimensional para la definición de AND y tabla de nodos equivalentes

En el ejemplo se añade primero la función local NAND[a,b; x1]. Al intentar añadir NAND[a,b; x2], como ya existe una entrada en la tabla con los índices a y b, la tabla nos devuelve x1 como salida, en lugar de agregar esta función redundante a la estructura; además se crea una entrada en la tabla de nodos equivalentes donde se indica que el nodo “x2” ha de ser reemplazado por el nodo “x1”.

Al ir a agregar la función NAND[x1,x2; c], utilizando la tabla de nodos equivalentes y reemplazando “x2” por “x1”, agregamos NAND[x1,x1; c].

Nuestra definición de la función queda por lo tanto:

AND[a,b; c]=

NAND[a,b; x1]

NAND[x1,x1; c]

3.3.2.3 *Ejemplo de funcionamiento del método de minimización aplicando las dos reglas*

Antes hemos definido el sumador de un bit como:

Sum[A,B,C_{in}; S,C_{out}] =

XOR[A,B; t1], AND[A,B; t2], XOR[A,B; t4]

AND[t1, C_{in}; t3], XOR[t4, C_{in}; S]

OR[t2,t3; C_{out}]

Donde es evidente que tenemos al menos una función redundante: dadas XOR[A,B; t1] y XOR[A,B; t4], t1 y t2 siempre tendrán el mismo valor.

3.3.2.3.1 Expresión como Red Booleana de funciones NAND

En primer lugar traducimos la función a funciones NAND.

Para ello sabemos que la expansión de AND en funciones NAND es:

AND[a,b; c]=

NAND[a,b; x1], NAND[a,b; x2]

NAND[x1,x2; c]

$$a + b = (a' + b')' = ((a\ a)'(b\ b)')'$$

Luego la expansión de OR en funciones NAND es:

OR[a,b; c]=

NAND[a,a; x1], NAND[b,b; x2]

NAND[x1,x2; c]

$$a \oplus b = (((a\ b)'a)'\ ((a\ b)'b)'\ )'$$

Luego la expansión de XOR en funciones NAND es:

XOR[a,b; c]=

NAND[a,b; x1] NAND[a,b; x2]

NAND[x1,a; x3],NAND[x2,b; x4]

NAND[x3,x4; c]

Aplicando la expansion de AND en funciones NAND obtenemos:

Sum[A,B,C_{in}; S,C_{out}]=

XOR[A,B; t1], NAND[A,B; x1], NAND[A,B; x2], XOR[A,B; t4]

NAND[t1, C_{in}; x3], NAND[t1, C_{in}; x4], XOR[t4, C_{in}; S] , NAND[x1,x2; t2]

NAND[x3, x4; t3],]

OR[t2,t3; C_{out}]

Aplicando la expansión de OR en funciones NAND obtenemos:

Sum[A,B,C_{in}; S,C_{out}]=

XOR[A,B; t1], NAND[A,B; x1], NAND[A,B; x2], XOR[A,B; t4]

NAND[t1, C_{in}; x3], NAND[t1, C_{in}; x4], XOR[t4, C_{in}; S] , NAND[x1,x2; t2]

NAND[t2,t2; x5], NAND[x3, x4; t3]

NAND{[t3,t3; x6]

NAND[x5,x6; C_{out}]

Aplicando la expansión de XOR en funciones NAND obtenemos:

Sum[A,B,C_{in}; S,C_{out}]=

NAND[A,B; x1], NAND[A,B; x2], NAND[A,B; x7], NAND[A,B; x8], NAND[A,B; x11], NAND[A,B; x12]

NAND[x7,A; x9],NAND[x8,B; x10] NAND[x11,A; x13],NAND[x12,B; x14];

NAND[x9,x10; t1], NAND[x13,x14; t4]

NAND[t1, C_{in}; x3], NAND[t1, C_{in}; x4], NAND[x1,x2; t2], NAND[t4, C_{in}; x15], NAND[t4, C_{in}; x16]

NAND[t2,t2; x5], NAND[x3, x4; t3], NAND[x15, t4; x17], NAND[x16, C_{in}; x18]

NAND{[t3,t3; x6], NAND[x17,x18; S]

NAND[x5,x6; C_{out}]

3.3.2.3.2 Segunda regla: funciones redundantes

Al usar una tabla dispersa bidimensional para aplicar la segunda regla y eliminar las funciones redundantes, obtendremos:

NodoEnL1	NodoEnL2	NodoSL
A	B	x1
A	x1	x9
B	x1	x10
x1	x1	t2
x9	x10	t1
C_{in}	t1	x3
t2	t2	x5
x3	x3	t3
x3	t1	x17
C_{in}	x3	x18
t3	t3	x6
x17	x18	S
x5	x6	C_{out}

Nodo	Nodo equivalente
x2	x1
x7	x1
x8	x1
x11	x1
x12	x1
x13	x9
x14	x10
t4	t1
x4	x3
x15	x3
x16	x3

Figura 10: Tabla dispersa bidimensional para la definición de Sum y tabla de nodos equivalentes

Sum[A,B,C_{in}; S,C_{out}]=

NAND[A,B; x1]

NAND[x1,A; x9],NAND[x1,B; x10],], NAND[x1,x1; t2]

NAND[x9,x10; t1]

NAND[t1, C_{in}; x3]

NAND[t2,t2; x5], NAND[x3, x3; t3], NAND[x3, t1; x17], NAND[x3, C_{in}; x18]

NAND{[t3,t3; x6], NAND[17,18; S]

NAND[x5,x6; C_{out}]

3.3.2.3.3 Primera regla: eliminación de la doble negación

Para cada nodo que vayamos añadiendo a la representación de la función, iremos comprobando que sus padres (los nodos de entrada de la función NAND de la que es salida) no son un mismo nodo, salida de otra negación. Si fuera el caso, estaríamos ante una doble negación y habría que eliminarla como en la Figura 7: Aplicación de la primera ley de De Morgan mediante una red de funciones NAND(c).

De esta forma ampliamos la tabla de nodos equivalentes con:

Nodo	Nodo equivalente
x5	x1
x6	x3

Figura 11: Nodos equivalentes por la doble negación

La definición de la función nos queda por lo tanto:

Sum[A,B,C_{in}; S,C_{out}]=

NAND[A,B; x1]

NAND[x1,A; x9],NAND[x1,B; x10]

NAND[x9,x10; t1]

NAND[t1, C_{in}; x3]

NAND[x3, t1; x17], NAND[x3, C_{in}; x18], NAND[x1,x3; C_{out}]

NAND[17,18; S]

La función inicial tenía **24** funciones NAND y una profundidad de **7** funciones NAND. Tras minimizarla, obtenemos una definición de la misma función compuesta por **9** funciones NAND y con una profundidad de **6**.

3.3.3 Complejidad computacional del método de minimización

El algoritmo consiste en ir añadiendo las funciones NAND, una a una, a una nueva estructura minimizada. Por lo tanto, el coste del algoritmo será proporcional al número de funciones booleanas que haya en la estructura a minimizar. Este coste lineal será el coste temporal de recorrer la estructura arborescente desde las hojas hasta las raíces.

3.3.3.1 *Complejidad de la eliminación de la doble negación*

Comprobar si se produce una doble negación tiene un coste constante, ya que solo hay que verificar si para el nodo salida de una función NAND las dos entradas son el mismo nodo (equivalente a una función NOT) y que este nodo no sea a su vez el resultado de una negación (véase la Figura 7: Aplicación de la primera ley de De Morgan mediante una red de funciones NAND).

3.3.3.1.1 Complejidad de eliminación de funciones redundantes

Para su implementación, necesitamos ordenar los nodos de forma que haya un único orden posible de las entradas de cada función NAND. Esto es, simplemente, una comparación entre dos nodos, operación de coste constante.

Mediante una tabla dispersa bidimensional (o equivalentemente una tabla dispersa de tablas dispersas), podemos comprobar si ya ha sido definida una función NAND con dos nodos concretos por entradas. El tiempo medio de acceso a una tabla dispersa bidimensional es constante.

3.3.3.1.2 Complejidad total

Teniendo en cuenta que el coste medio de las operaciones a realizar por nodo añadido es constante, y que el coste del algoritmo será proporcional al número de nodos de salida de funciones añadidos, el coste total medio será por lo tanto lineal en el número de nodos de salida de funciones añadidos. Esto es equivalente al número de funciones NAND. En notación asintótica tenemos un coste medio *O(n)*.

3.3.4 Limitaciones del método de minimización

3.3.4.1 *Funciones de dos entradas*

Usar únicamente funciones NAND de dos entradas es una decisión de diseño, ya que permite simplificar las expresiones con gran facilidad.

Mediante funciones con más entradas se pueden expresar de forma más compacta expresiones arbitrarias. Un ejemplo de esta posibilidad, llevado al límite, es la minimización en dos niveles: se realiza una única suma de una cantidad arbitraria de productos. Sin embargo, la selección del número de entradas de las funciones entra dentro de la optimización dependiente de la tecnología, por lo que no parece relevante el número de entradas de las funciones en una expresión intermedia.

3.3.4.2 *Funciones equivalentes irreducibles*

Hay funciones que pueden representarse de diferentes maneras mediante funciones NAND y todas son irreducibles por este método.

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

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

$$A \oplus B = (((A\,B)'A)'\ \ ((A\,B)'B)'\ )'$$

Estas representaciones dependen exclusivamente de cómo se defina la función y tienen costes distintos.

Así la primera expresión consta de 6 funciones NAND diferentes y una profundidad de 4.

La segunda expresión tiene 5 funciones diferentes y una profundidad de 3.

La última expresión tiene 4 funciones diferentes y una profundidad de 3.

Esto sugiere, en primer lugar, que la optimización consiste en algo más que eliminar la redundancia, al menos al expresar algoritmos secuenciales a nivel lógico como funciones NAND. En segundo lugar, sugiere que nos encontramos ante un método de minimización heurístico, ya que el resultado de minimizar una función depende de cómo se defina ésta, y no va a hallar necesariamente la minimización con menor cantidad de funciones, ni menor profundidad.

3.4 Optimización de algoritmos secuenciales

En el caso de un algoritmo secuencial donde todos los nodos son de igual tamaño, habrá que traducir el algoritmo a funciones NAND y usar el método de minimización. El método de minimización nos devolverá una tabla de nodos equivalentes que aplicaremos a la función original para optimizarla. Por último, se ejecutará un método de reconstrucción de la función para asegurar que solo contiene los nodos utilizados.

Primero introduciremos el algoritmo en nuestra base de datos y luego usaremos esta definición para traducirlo a funciones NAND.

3.4.1 Traducción a funciones NAND mediante la base de datos

Para traducir una función definida mediante la base de datos a funciones NAND, tan sólo hay que expandir recursivamente las definiciones instanciadas. El coste de acceso a la base de datos es constante al usar una tabla dispersa, por lo tanto el coste total de la expansión será lineal en el número de instancias.

Por ejemplo, si tenemos la siguiente base de datos:

```
BASE DE DATOS:
not[0; 1] =
        nand [0,0; 1]
nand[0,1; 2] =
        nand [0,1; 2]
or[0,1; 2] =
        not [0; 3]  not [1; 4]
        nand [3,4; 2]
```

Para expandir la función OR en funciones NAND, expandiremos sus instancias, consultando la base de datos en busca de la definición instanciada, quedando la función:

```
or[0,1; 2] =
        nand[0,0; 3]  nand [1,1; 4]
        nand [3,4; 2]
```

Dado que traducir una función a funciones NAND tiene coste lineal y el método de minimización tiene un coste medio lineal, el coste medio de optimizar algoritmos secuenciales mediante este método será también lineal.

3.4.2 Método de reconstrucción de funciones

Este método, primero, borra las referencias a todos los nodos e instancias de la función.

Después, recorre la función desde los nodos de salida hasta las entradas, pasando por todas las instancias y añade los nodos e instancias recorridos.

La utilidad del método es que no queden nodos “muertos” a causa de simplificaciones, y que puedan seguir apareciendo como pertenecientes a la función.

3.5 Optimización de la parte de secuencial de algoritmos recursivos

El primer paso será eliminar temporalmente la recursividad, para extraer únicamente la parte secuencial.

Para ello, añadimos los nodos de entrada de funciones recursivas como nodos de salida y los nodos de salida de funciones recursivas como nodos de entrada, y eliminamos la llamada recursiva.

Sea, por ejemplo, la función recursiva:

```
nandR[0,1; 2]=
        nand[0,1{n}; 2{n}]
        nandR[0,1{1..n-1}; 2{1..n-1}]
```

La transformaremos temporalmente en:

```
nandR[0,1, 2{1..n-1}]; 2,0,1{1..n-1}]=
        nand[0,1{n}; 2{n}]
```

Optimizaremos como en la sección 3.4, tratando la función como si se tratara de un algoritmo secuencial.

Por último, desharemos la transformación para volver a tener un algoritmo recursivo.

Además, un algoritmo puede tener nodos de diferentes tamaños, por lo que habrá que dividirlos para que tengan el mismo tamaño. En consecuencia, haremos una fisión de los nodos antes de transformar el algoritmo recursivo, y los fusionaremos tras deshacer la transformación.

3.5.1 Algoritmo de fisión de nodos

El algoritmito de fisión de nodos es un algoritmo recursivo. Parte de los nodos de salida y acaba en los nodos de entrada.

Si un nodo es salida de una instancia de función NAND y tiene hijos, los hijos se convierten en salida de una nueva función NAND cuyas entradas son los hijos de las entradas de la instancia original.

3.5.2 Algoritmo de fusión de nodos

El algoritmo de fusión hace la operación inversa. Si hay tres nodos (dos de entrada y uno de salida) cuyos subnodos forman funciones NAND, los fusiona en una sola.

Como en el caso del algoritmo de fisión, es recursivo desde las salidas hasta las entradas.

3.6 Optimización de la redundancia en algoritmos recursivos

El método anterior no contempla la redundancia que puede darse entre llamadas recursivas de un algoritmo. Para optimizar la redundancia entre llamadas recursivas sucesivas de una función, necesitaremos desplegarlas.

Dado que el método de minimización utilizado sólo elimina la doble negación y las funciones repetidas, será suficiente con desplegar la llamada recursiva una sola vez.

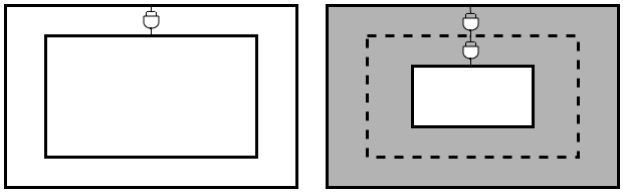


Figura 12: Representación del desplegado de una llamada recursiva

Si imaginamos una función recursiva como una caja (Figura 12), vemos que al desplegarla una sola vez se producirán todas las dobles negaciones que una función recursiva pueda producir consigo misma. Las funciones repetidas también aparecerán de la misma manera.

Para hallar la función recursiva simplificada, se almacena qué nodos están definidos antes de expandir y a qué nodos de la función original corresponden los nodos expandidos.

Después de desplegar la recursividad, aplicaremos el método de minimización como en la sección 3.5.

Si hay nodos equivalentes, crearemos una nueva función recursiva.

Las entradas de la nueva función recursiva serán nodos originales de la función antes de expandir que sean entrada de una instancia de función con un nodo de salida nuevo.

Las salidas de la nueva función recursiva serán nodos originales de la función que sean salida de una instancia con un nodo nuevo como entrada.

Localizando los nodos correspondientes en la función original a las entradas de la función expandida, tendremos los nodos de entrada de la instancia a la nueva función recursiva.

Localizando los nodos correspondientes en la función original a las salidas de la función expandida, tendremos los nodos de salida de la instancia a la nueva función recursiva.

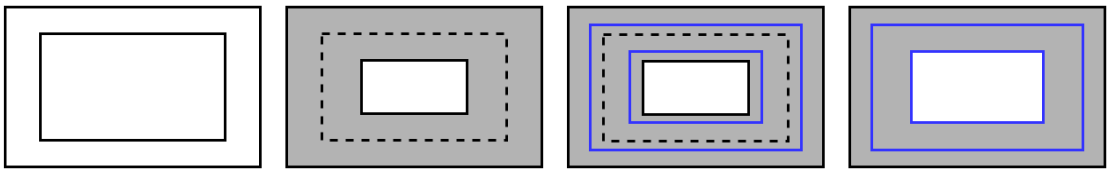


Figura 13: Optimización de una función recursiva

En la Figura 13 vemos este procedimiento. En primer lugar tenemos el desplegado de la función recursiva. En segundo lugar se halla la nueva función recursiva (en azul en el dibujo), eliminando la redundancia.

Hay que tener en cuenta, que si hallamos todos los nodos *hijos* de un mismo nodo *padre*, será necesario reemplazarlos por este nodo *padre*. A este procedimiento lo llamaremos *fusión* de nodos.

Sea, por ejemplo, la función recursiva:

fooR[0,1; 2,3]=

nand[0,1{1..n-1}{n}; 2{1..n-1}{n},3], nand[0,1{n}; 2{n}]

fooR[0,1{1..n-1}; 2{1..n-1},4]

Si expandimos una vez la llamada recursiva, obtenemos:

fooR[0,1; 2,3]=

nand[0,1{1..n-1}{n}; 2{1..n-1}{n},3], nand[0,1{n}; 2{n}]

nand[0,1{1..n-1}{1..n-1}{n}; 2{1..n-1}{1..n-1}{n},4], nand[0,1{1..n-1}{n}; 2{1..n-1}{n}]

fooR[0,1{1..n-1}{1..n-1}; 2{1..n-1}{1..n-1},5]

Donde, mediante el método de minimización, hallamos que los nodos 3 y 2{1..n-1}{n} son equivalentes.

Los nodos originales antes de la expansión son:

0,1,2,3, 1{1..n-1}{n},2{1..n-1}{n},1{1..n-1}; 2{1..n-1},1{n},2{n},4

0	0
1{1..n-1}{1..n-1}{n}	1{1..n-1}{n}
2{1..n-1}{1..n-1}{n}	2{1..n-1}{n}
4	3
1{1..n-1}{n}	1{n}
2{1..n-1}{n}	2{n}
1{1..n-1}{1..n-1}	1{1..n-1}
2{1..n-1}{1..n-1}	2{1..n-1}
1{1..n-1}	1
2{1..n-1}	2

Figura 14: Tabla de relación de nodos expandidos con nodos de la función original.

A partir de los nodos de salida, encontramos los que serán nodos de salida de una nueva función recursiva. Estos serán los nodos que sean originales de la función antes de la expansión y que sean salida de una función con entradas nuevas a causa de la expansión.

En el ejemplo tendríamos los nodos de salida de la nueva función recursiva:

2{1..n-1}{1..n-1}, 2{1..n-1}{n}

Si los fusionamos, serían:

2{1..n-1}

Y los nodos de entrada de la nueva función recursiva:

0, 1{1..n-1}{1..n-1},1{1..n-1}{n}

Si los fusionamos, serían:

0, 1{1..n-1}

Mediante la tabla encontramos los nodos de salida:

2{1..n-1}, 2{n}

Si los fusionamos, serían:

2

Y los nodos de entrada:

0, 1{1..n-1},1{n}

Si los fusionamos, serían:

0,1

Con estos nodos la función original nos queda:

fooR[0,1; 2,3]=

nand[0,1{1..n-1}{n}; 2{1..n-1}{n},3]

newFooR[0,1; 2]

Y la nueva función recursiva:

newFooR[0,1; 2]=

nand[0,1{n}; 2{n}]

newFooR[0,1{1..n-1}; 2{1..n-1}]

3.7 Algoritmos que contienen una llamada recursiva

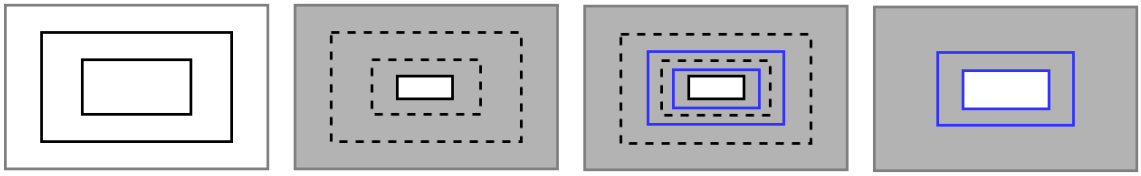


Figura 15: Optimización de una función que contiene una llamada recursiva

En este caso se optimizará primero la parte secuencial como se describió en la sección 3.5.

Después, se expandirá la función recursiva como en la sección 3.6. Si el método de minimización no encuentra nodos equivalentes, se dejará la función como está. En caso contrario se calculará la nueva función recursiva.

Finalmente, se volverá a optimizar la parte secuencial, ya que esta ha podido crecer a causa del método de eliminación de redundancia en algoritmos recursivos.

3.8 Evaluación de funciones/Ejecución de algoritmos

Como ya se ha mencionado previamente, utilizamos funciones recursivas para representar algoritmos, como por ejemplo el de la suma.

A fin de comprobar los resultados obtenidos durante el proyecto, se ha desarrollado un método de evaluación de estas funciones. Esto es equivalente a la ejecución de algoritmos.

La evaluación se realiza en profundidad, mediante llamadas recursivas.

Se conserva en todo momento una tabla que relaciona las variables/los nodos con valores concretos y se evalúan recursivamente a partir de las salidas hasta las entradas. Siempre se evalúan primero los nodos “final” y después el “resto”.

Por ejemplo, si queremos evaluar la función:

not[0; 1]=

nand[0,0; 1]

Empezaremos por asignarle valores a las entradas:

Nodo	Valor
0	‘1’

Figura 16: Tabla de valores de nodos antes de ejecución

Después, se empezará a evaluar de forma recursiva desde las salidas.

Al evaluar la instancia “nand[0,0; 1]”, como las entradas ya están definidas, podremos calcular la salida. Si no fuera el caso se evaluarían recursivamente las entradas.

Por lo tanto, obtenemos la tabla:

Nodo	Valor
0	‘1’
1	‘0’

Figura 17: Tabla de valores de nodos después de ejecución

Con esto ya hemos ejecutado la función y obtenemos el valor “0”.

4 Pruebas y resultados obtenidos

Para evaluar los resultados obtenidos, los compararemos con los que proporciona una herramienta de minimización de circuitos.

Se ha elegido el software Logic Friday [14] por su simplicidad y porque permite usar los algoritmos de Quine–McCluskey, Espresso y Espresso-exacto en cuanto a minimización de dos niveles y MisII en cuanto a minimización multinivel. Estos algoritmos se han mencionado en mayor detalle en la sección 3.3.

Nos interesa especialmente el algoritmo MisII, ya que permite minimizar circuitos secuenciales compuestos únicamente de puertas lógicas NAND de dos entradas.

Dado que este proyecto optimiza funciones recursivas, tendremos que evaluar casos particulares para poder comparar los resultados.

Logic Friday tiene su propia notación para expresar circuitos, que detallamos a continuación:

&, * o juxtaposición	AND
| o +	OR
! (prefijo) o '(sufijo)	complemento
!= o ^	XOR
==	NOR

Figura 18: Sintaxis de Logic Friday

4.1 Minimización de XOR

Definimos:

$$A \oplus B = (((A\,B)'A)'\ ((A\,B)'B)'\ )'$$

4.1.1 Función minimizada mediante nuestro método

Notación propia:

xor[0,1; 2] =

nand [0,1; 3]

nand [3,0; 4], nand [3,1; 5]

nand [4,5; 2]

Este método no es capaz de minimizar más esta expresión.

4.1.1.1 *Coste de la función minimizada mediante nuestro método*

Este circuito utiliza **4** funciones NAND y tiene una profundidad de **3** funciones.

4.1.2 Función minimizada mediante el algoritmo de Quine–McCluskey

Si elegimos la opción de minimización exacta de Logic Friday, estamos utilizando el algoritmo de Quine–McCluskey, que nos da la minimización en dos niveles.

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

4.1.3 Función minimizada mediante el algoritmo MisII

Mediante la opción de minimización de circuitos de Logic Friday y seleccionando como únicos componentes puertas lógicas NAND de 2 entradas, obtenemos el siguiente circuito:

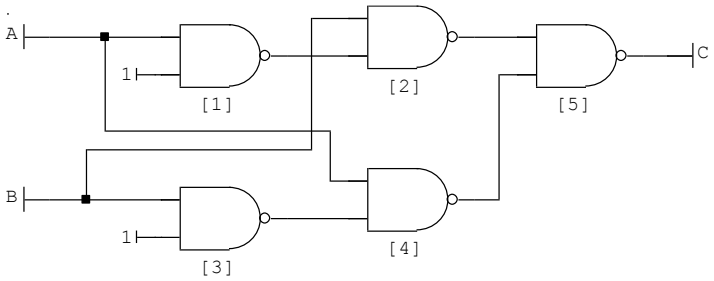


Figura 19: Función XOR minimizada mediante MisII

4.1.3.1 *Coste de la función minimizada mediante MisII*

Este circuito utiliza **5** funciones NAND y tiene una profundidad de **3** funciones NAND, además tiene un coste secundario por el uso de dos constantes en las entradas.

4.2 Multiplexor de dos entradas (if-then-else lógico)

Definimos:

$$Z = (A\,S') + (B\,S)$$

4.2.1 Función minimizada mediante nuestro método

Notación propia:

if[0,1,2; 6] =

nand [0,0; 3] nand [0,2; 5]

nand [3,1; 4]

nand [4,5; 6]

4.2.1.1 *Coste de la función minimizada mediante nuestro método*

Utiliza **4** funciones NAND y tiene una profundidad de **3** funciones.

4.2.2 Función minimizada mediante el algoritmo de Quine–McCluskey

$$Z = (A\,S') + (B\,S)$$

4.2.3 Función minimizada mediante el algoritmo MisII

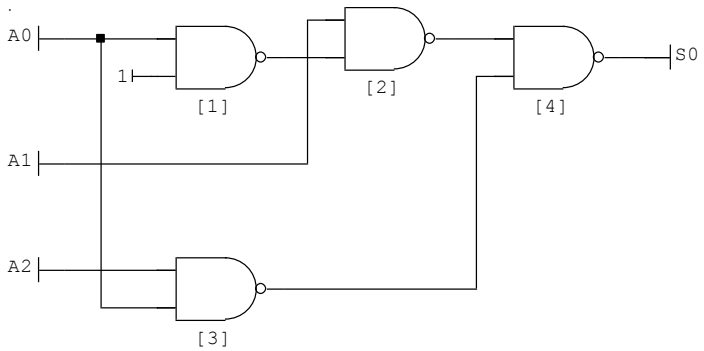


Figura 20: Multiplexor minimizado mediante MisII

4.2.3.1 *Coste de la función minimizada mediante MisII*

Utiliza **4** funciones NAND y tiene una profundidad de **3** funciones; además tiene un coste secundario por el uso de una constante en la entrada.

4.3 Expresión general de la suma.

Podemos obtener el algoritmo general de la suma utilizando, de forma recursiva, el sumador completo que se ha definido en la sección 3.1.

add[0,1; 10(8{n}&7&11)] =

and [0{n},1{n}; 6] xor [0{n},1{n}; 11]

sumR [0{1..n-1},1{1..n-1},6; 7,8]

sumR[0,1,2; 3(5&6),4(7&8)] =

sum [0{n},1{n},2; 6,7]

sumR [0{1..n-1},1{1..n-1},7; 5,8]

sum[0,1,2; 10,11] =

xor[0,1; t1], and[0,1; t2]

and[t1,2; t3], xor[t1,C; 10]

or[t2,t3; 11]

Que expresado en funciones NAND y minimizado queda:

add[0,1; 10(8{n}&7&11)] =

nand [0{n},1{n}; 12]

nand [12,12; 6] nand [12,0{n}; 14] nand [12,1{n}; 15]

sumR [0{1..n-1},1{1..n-1},6; 7,8] nand [14,15; 11]

sumR[0,1,2; 3(5&6),4(7&8)] =

sum [0{n},1{n},2; 6,7]

sumR [0{1..n-1},1{1..n-1},7; 5,8]

sum[0,1,2; 10,11] =

nand [0,1; 3]

nand [3,0; 4] nand [3,1; 5]

nand [4,5; 6]

nand [6,2; 7]

nand [7,6; 8] nand [7,2; 9] nand [3,7; 11]

nand [8,9; 10]

Observamos que en la parte secuencial de la definición tenemos 5 funciones NAND de dos bits de entrada.

Para la parte recursiva, el número de recursiones es igual al de bits de los operandos menos uno, ya que en cada recursión obtenemos un bit de salida y el acarreo, además del bit de salida y el acarreo obtenido en la parte secuencial. En cada recursión tenemos 9 funciones NAND de dos bits de entrada.

Por lo tanto, sea *n* el número de bits de los operandos, el **número de funciones** será:

$$5 + 9(n - 1)$$

Para calcular una **cota** de la profundidad de la suma en función del número de bits n de los operandos, basta con observar que hay una profundidad constante que corresponde a la parte recursiva y una profundidad variable que corresponde a la parte recursiva de la definición, y depende directamente de la cantidad de bits.

Por lo tanto, la **profundidad** es, como **máximo**:

$$2 + 6n$$

4.3.1 Caso particular de la suma: ADD de 3 bits

4.3.1.1 *Función minimizada mediante nuestro método*

A partir de la expresión general de la suma, desenrollando la función recursiva dos veces, obtenemos:

add3bits[0,1; 28&38(37&41)&45)] =

nand [0{1..n-1}{n},1{1..n-1}{n}; 10] nand [0{n},1{n}; 14] nand [0{1..n-1}{1..n-1}{n},1{1..n-1}{1..n-1}{n}; 23]

nand [10,0{1..n-1}{n}; 11] nand [10,1{1..n-1}{n}; 12] nand [14,14; 15] nand [23,0{1..n-1}{1..n-1}{n}; 24] nand [23,1{1..n-1}{1..n-1}{n}; 25] nand [14,0{n}; 43] nand [14,1{n}; 44]

nand [11,12; 13] nand [24,25; 26] nand [43,44; 45]

nand [13,15; 16]

nand [10,16; 17] nand [16,13; 39] nand [16,15; 40]

nand [26,17; 27] nand [39,40; 41]

nand [23,27; 28] nand [27,26; 35] nand [27,17; 36]

nand [35,36; 37]

Hemos implementado un método para transformar automáticamente la notación propia a notación de Logic Friday:

```
S3=!(!(A2 B2) !(!(!(!(A2 B2) A2) !(!(A2 B2) B2)) !(!(A1 B1)
!(!(!(!(A1 B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0))))));

S2=!(!(!(!(!(!(A2 B2) A2) !(!(A2 B2) B2)) !(!(A1 B1) !(!(!(!(A1
B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0))))) !(!(!(A2 B2) A2)
!(!(A2 B2) B2))) !(!(!(!(!(A2 B2) A2) !(!(A2 B2) B2)) !(!(A1 B1)
!(!(!(!(A1 B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0))))) !(!(A1
B1) !(!(!(!(A1 B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0))))));

S1=!(!(!(!(!(!(!(A1 B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0)))
!(!(!(A1 B1) A1) !(!(A1 B1) B1))) !(!(!(!(!(A1 B1) A1) !(!(A1 B1)
B1)) !(!(A0 B0) !(A0 B0))) !(!(A0 B0) !(A0 B0))));

S0=!(!(!(A0 B0) A0) !(!(A0 B0) B0));
```

S0-S3 representan los bits de salida desde el de menor peso hasta el de mayor peso.

A0-A2 representan los 3 bits de entrada del primer operando, desde el de menor peso hasta el de mayor peso.

B0-B2 representan los 3 bits de entrada del segundo operando, desde el de menor peso hasta el de mayor peso. Comprobamos que la tabla de verdad es efectivamente la de la suma de 3 bits:

A2	B2	A1	B1	A0	B0		S3	S2	S1	S0
0	0	0	0	0	0		0	0	0	0
0	0	0	0	0	1		0	0	0	1
0	0	0	0	1	0		0	0	0	1
0	0	0	0	1	1		0	0	1	0
0	0	0	1	0	0		0	0	1	0
0	0	0	1	0	1		0	0	1	1
0	0	0	1	1	0		0	0	1	1
0	0	0	1	1	1		0	1	0	0
0	0	1	0	0	0		0	0	1	0
0	0	1	0	0	1		0	0	1	1
0	0	1	0	1	0		0	0	1	1
0	0	1	0	1	1		0	1	0	0
0	0	1	1	0	0		0	1	0	0
0	0	1	1	0	1		0	1	0	1
0	0	1	1	1	0		0	1	0	1
0	0	1	1	1	1		0	1	1	0
0	1	0	0	0	0		0	1	0	0
0	1	0	0	0	1		0	1	0	1
0	1	0	0	1	0		0	1	0	1
0	1	0	0	1	1		0	1	1	0
0	1	0	1	0	0		0	1	1	0
0	1	0	1	0	1		0	1	1	1
0	1	0	1	1	0		0	1	1	1
0	1	0	1	1	1		1	0	0	0
0	1	1	0	0	0		0	1	1	0
0	1	1	0	0	1		0	1	1	1
0	1	1	0	1	0		0	1	1	1
0	1	1	0	1	1		1	0	0	0
0	1	1	1	0	0		1	0	0	0
0	1	1	1	0	1		1	0	0	1
0	1	1	1	1	0		1	0	0	1
0	1	1	1	1	1		1	0	1	0
1	0	0	0	0	0		0	1	0	0
1	0	0	0	0	1		0	1	0	1
1	0	0	0	1	0		0	1	0	1
1	0	0	0	1	1		0	1	1	0
1	0	0	1	0	0		0	1	1	0
1	0	0	1	0	1		0	1	1	1
1	0	0	1	1	0		0	1	1	1
1	0	0	1	1	1		1	0	0	0
1	0	1	0	0	0		0	1	1	0
1	0	1	0	0	1		0	1	1	1
1	0	1	0	1	0		0	1	1	1
1	0	1	0	1	1		1	0	0	0
1	0	1	1	0	0		1	0	0	0
1	0	1	1	0	1		1	0	0	1

1	0	1	1	1	0		1	0	0	1
1	0	1	1	1	1		1	0	1	0
1	1	0	0	0	0		1	0	0	0
1	1	0	0	0	1		1	0	0	1
1	1	0	0	1	0		1	0	0	1
1	1	0	0	1	1		1	0	1	0
1	1	0	1	0	0		1	0	1	0
1	1	0	1	0	1		1	0	1	1
1	1	0	1	1	0		1	0	1	1
1	1	0	1	1	1		1	1	0	0
1	1	1	0	0	0		1	0	1	0
1	1	1	0	0	1		1	0	1	1
1	1	1	0	1	0		1	0	1	1
1	1	1	0	1	1		1	1	0	0
1	1	1	1	0	0		1	1	0	0
1	1	1	1	0	1		1	1	0	1
1	1	1	1	1	0		1	1	0	1
1	1	1	1	1	1		1	1	1	0

4.3.1.1.1 Coste de la función minimizada mediante nuestro método

La suma de 3 bits, minimizada mediante nuestro método, usa **23** funciones NAND y tiene una profundidad de **8** funciones.

4.3.1.2 *Función minimizada mediante el algoritmo de Quine–McCluskey*

Minimizando mediante Logic Friday con la opción “exacta” para usar el método de Quine-McCluskey, y con la opción de minimizar las ecuaciones de forma conjunta para obtener el mínimo posible de productos, obtenemos:

```
Minimized:
S3 = B2 B1 A0 B0 + A2 B1 A0 B0 + B2 A1 A0 B0 + A2 A1 A0 B0 + B2
A1 B1  + A2 A1 B1  + A2 B2 ;
S2 = A2' B2' B1 A0 B0 + A2' B2' A1 A0 B0 + A2 B2 B1 A0 B0 + A2 B2
A1 A0 B0 + A2 B2' B1' B0' + A2' B2 B1' B0' + A2 B2' A1' B0' + A2'
B2 A1' B0' + A2 B2' B1' A0'  + A2' B2 B1' A0'  + A2 B2' A1' A0'  +
A2' B2 A1' A0'  + A2 B2' A1' B1'  + A2' B2 A1' B1'  + A2' B2' A1
B1  + A2 B2 A1 B1 ;
S1 = A1' B1' A0 B0 + A1 B1 A0 B0 + A1 B1' B0' + A1' B1 B0' + A1
B1' A0'  + A1' B1 A0' ;
S0 = A0 B0' + A0' B0;
```

4.3.1.3 *Función minimizada mediante el algoritmo MisII*

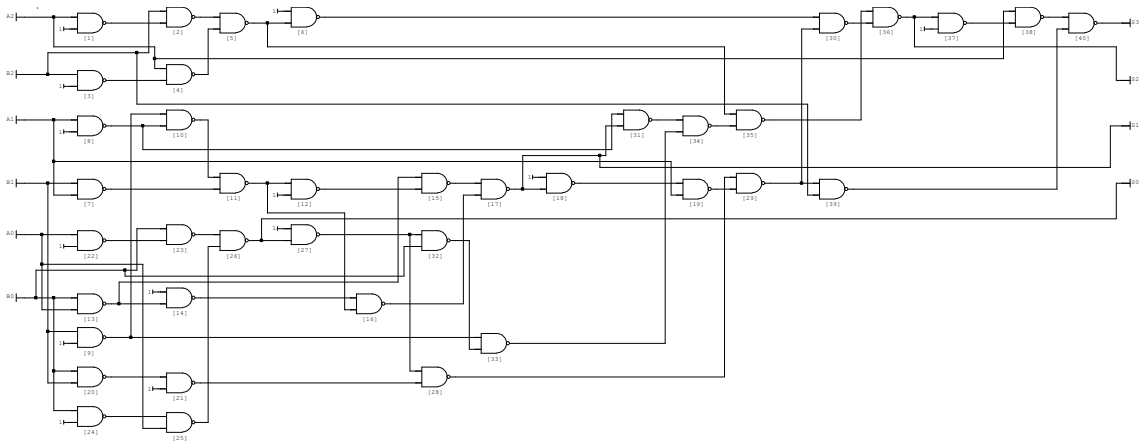


Figura 21: Función suma de 3 bits minimizada mediante MisII

4.3.1.3.1 Coste de la función minimizada mediante MisII

Utiliza **40** funciones NAND y tiene una profundidad de **13** funciones.

	Nuestro método	MisII
Funciones totales	23	40
Profundidad	8	13

Tabla 1 Evaluación de la minimización de la suma de 3 bits

4.3.2 Caso particular de la suma: ADD de 4 bits

4.3.2.1 *Función minimizada mediante nuestro método*

A partir de la expresión general de la suma, desenrollando la función recursiva tres veces, obtenemos:

```
add4bits[0,1; 39&53(49(48&52)&56)&60)] =
      nand [0{1..n-1}{n},1{1..n-1}{n}; 10] nand [0{n},1{n}; 14] nand
[0{1..n-1}{1..n-1}{n},1{1..n-1}{1..n-1}{n}; 23] nand [0{1..n-1}{1..n-
1}{1..n-1}{n},1{1..n-1}{1..n-1}{1..n-1}{n}; 34]
      nand [10,0{1..n-1}{n}; 11] nand [10,1{1..n-1}{n}; 12] nand [14,14;
15] nand [23,0{1..n-1}{1..n-1}{n}; 24] nand [23,1{1..n-1}{1..n-1}{n};
25] nand [34,0{1..n-1}{1..n-1}{1..n-1}{n}; 35] nand [34,1{1..n-1}{1..n-
1}{1..n-1}{n}; 36] nand [14,0{n}; 58] nand [14,1{n}; 59]
      nand [11,12; 13] nand [24,25; 26] nand [35,36; 37] nand [58,59;
60]
      nand [13,15; 16]
      nand [10,16; 17] nand [16,13; 54] nand [16,15; 55]
      nand [26,17; 27] nand [54,55; 56]
      nand [23,27; 28] nand [27,26; 50] nand [27,17; 51]
      nand [37,28; 38] nand [50,51; 52]
      nand [34,38; 39] nand [38,37; 46] nand [38,28; 47]
      nand [46,47; 48]
```

Transformando la expresión a la notación de Logic Friday:

```
S4=!(!(A3 B3) !(!(!(!(A3 B3) A3) !(!(A3 B3) B3)) !(!(A2 B2)
!(!(!(!(A2 B2) A2) !(!(A2 B2) B2)) !(!(A1 B1) !(!(!(!(A1 B1) A1)
!(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0)))))))));
S3=!(!(!(!(!(!(A3 B3) A3) !(!(A3 B3) B3)) !(!(A2 B2) !(!(!(!(A2
B2) A2) !(!(A2 B2) B2)) !(!(A1 B1) !(!(!(!(A1 B1) A1) !(!(A1 B1)
B1)) !(!(A0 B0) !(A0 B0))))))) !(!(!(A3 B3) A3) !(!(A3 B3) B3)))
!(!(!(!(!(!(A3 B3) A3) !(!(A3 B3) B3)) !(!(A2 B2) !(!(!(!(A2 B2) A2)
!(!(A2 B2) B2)) !(!(A1 B1) !(!(!(!(A1 B1) A1) !(!(A1 B1) B1))
!(!(A0 B0) !(A0 B0))))))) !(!(A2 B2) !(!(!(!(A2 B2) A2) !(!(A2 B2)
B2)) !(!(A1 B1) !(!(!(!(A1 B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0
B0)))))))));
S2=!(!(!(!(!(!(A2 B2) A2) !(!(A2 B2) B2)) !(!(A1 B1) !(!(!(!(A1
B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0))))) !(!(!(A2 B2) A2)
!(!(A2 B2) B2))) !(!(!(!(!(A2 B2) A2) !(!(A2 B2) B2)) !(!(A1 B1)
!(!(!(!(A1 B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0))))) !(!(A1
B1) !(!(!(!(A1 B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0))))));
S1=!(!(!(!(!(!(A1 B1) A1) !(!(A1 B1) B1)) !(!(A0 B0) !(A0 B0)))
!(!(!(A1 B1) A1) !(!(A1 B1) B1))) !(!(!(!(!(A1 B1) A1) !(!(A1 B1)
B1)) !(!(A0 B0) !(A0 B0))) !(!(A0 B0) !(A0 B0))))
S0=!(!(!(A0 B0) A0) !(!(A0 B0) B0));
```

Comprobamos que la tabla de verdad de la sección 7.4 es efectivamente la de la suma de 4 bits.

4.3.2.1.1 Coste de la función minimizada mediante nuestro método

La suma de 4 bits, minimizada mediante nuestro método, usa **32** funciones NAND y tiene una profundidad de **10** funciones.

4.3.2.2 *Función minimizada mediante el algoritmo de Quine-McCluskey*

Minimizando mediante Logic Friday con la opción "exacta" para usar el método de Quine-McCluskey, y con la opción de minimizar las ecuaciones de forma conjunta para obtener el mínimo posible de productos, obtenemos:

```
Minimized:
S4 = B3 B2 B1 A0 B0 + A3 B2 B1 A0 B0 + B3 A2 B1 A0 B0 + A3 A2 B1
A0 B0 + B3 B2 A1 A0 B0 + A3 B2 A1 A0 B0 + B3 A2 A1 A0 B0 + A3 A2
A1 A0 B0 + B3 B2 A1 B1  + A3 B2 A1 B1  + B3 A2 A1 B1  + A3 A2 A1
B1  + B3 A2 B2  + A3 A2 B2  + A3 B3 ;
S3 = A3' B3' B2 B1 A0 B0 + A3' B3' A2 B1 A0 B0 + A3' B3' B2 A1 A0
B0 + A3' B3' A2 A1 A0 B0 + A3 B3 B2 B1 A0 B0 + A3 B3 A2 B1 A0 B0
+ A3 B3 B2 A1 A0 B0 + A3 B3 A2 A1 A0 B0 + A3 B3' B2' B1' B0' + A3'
B3 B2' B1' B0' + A3 B3' A2' B1' B0' + A3' B3 A2' B1' B0' + A3 B3'
B2' A1' B0' + A3' B3 B2' A1' B0' + A3 B3' A2' A1' B0' + A3' B3 A2'
A1' B0' + A3 B3' B2' B1' A0'  + A3' B3 B2' B1' A0'  + A3 B3' A2'
B1' A0'  + A3' B3 A2' B1' A0'  + A3 B3' B2' A1' A0'  + A3' B3 B2'
A1' A0'  + A3 B3' A2' A1' A0'  + A3' B3 A2' A1' A0'  + A3 B3' B2'
A1' B1'  + A3' B3 B2' A1' B1'  + A3 B3' A2' A1' B1'  + A3' B3 A2'
A1' B1'  + A3' B3' B2 A1 B1  + A3' B3' A2 A1 B1  + A3 B3 B2 A1 B1
+ A3 B3 A2 A1 B1  + A3 B3' A2' B2'  + A3' B3 A2' B2'  + A3' B3'
A2 B2  + A3 B3 A2 B2 ;
S2 = A2' B2' B1 A0 B0 + A2' B2' A1 A0 B0 + A2 B2 B1 A0 B0 + A2 B2
A1 A0 B0 + A2 B2' B1' B0' + A2' B2 B1' B0' + A2 B2' A1' B0' + A2'
B2 A1' B0' + A2 B2' B1' A0'  + A2' B2 B1' A0'  + A2 B2' A1' A0'  +
A2' B2 A1' A0'  + A2 B2' A1' B1'  + A2' B2 A1' B1'  + A2' B2' A1
B1  + A2 B2 A1 B1 ;
S1 = A1' B1' A0 B0 + A1 B1 A0 B0 + A1 B1' B0' + A1' B1 B0' + A1
B1' A0'  + A1' B1 A0' ;
S0 = A0 B0' + A0' B0;
```

4.3.2.3 *Función minimizada mediante el algoritmo MisII*

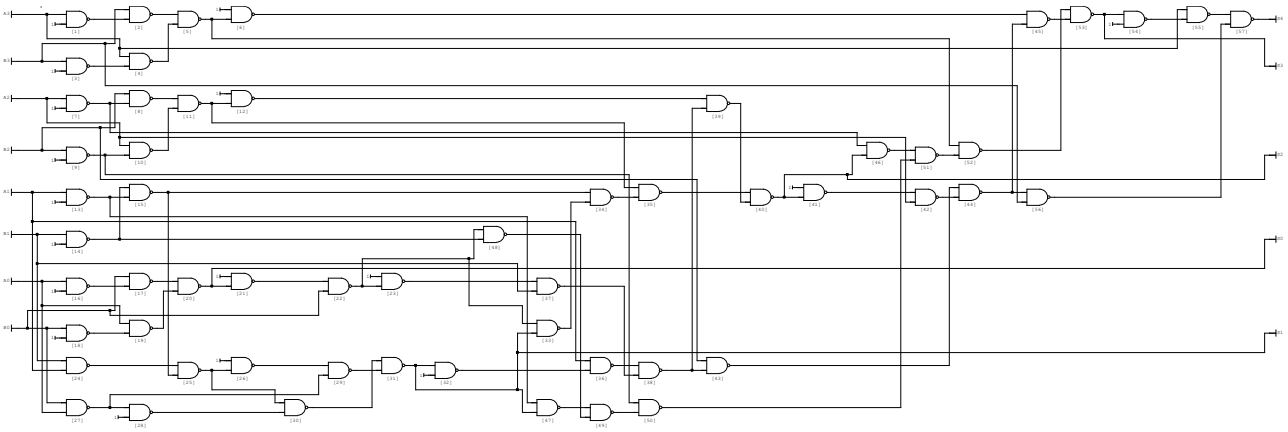


Figura 22: Función suma de 4 bits minimizada mediante MisII

4.3.2.3.1 Coste de la función minimizada mediante MisII

Utiliza **57** funciones NAND y tiene una profundidad de **18** funciones.

	Nuestro método	MisII
Funciones totales	32	57
Profundidad	10	18

Tabla 2: evaluación de la minimización de la suma de 4 bits

5 Consideraciones finales

5.1 Conclusiones

En todos los ejemplos estudiados, nuestro método minimiza mucho mejor que MisII para el caso concreto de circuitos compuestos de funciones NAND de dos entradas. Cuanto más complejas son las funciones minimizadas, más se aprecia esta mejora.

MisII es un algoritmo más flexible, ya que permite elegir qué puertas lógicas y de qué tamaño podemos usar para construir el circuito minimizado.

Sin embargo, es importante destacar que lo que lo que ha optimizado nuestro método es el algoritmo general de la suma, mientras que MisII ha optimizado circuitos concretos. Esto debería de resultar una ventaja para MisII, pero en la práctica no lo ha sido.

Estos resultados apoyan la tesis de que el método de minimización aquí presentado tiene un buen comportamiento y escala particularmente bien, además de ser muy rápido.

Dado que en 2008 se demostró que el problema de minimización de circuitos para funciones booleanas es $\sum_2^P$ −completo [3], el método aquí propuesto parece ser una buena heurística.

Cabe mencionar la relevancia de la profundidad de los circuitos obtenidos, porque en una arquitectura paralela es el factor determinante de su velocidad. En el caso de la suma de 4 bits la profundidad hallada por MisII es un 80% superior a la que encuentra nuestro método.

Sería interesante realizar un análisis en mayor profundidad de la eficacia de la minimización, pero eso queda fuera del alcance de este proyecto.

El enfoque utilizado para optimizar en este proyecto parece tener un gran potencial.

La únicas problemáticas posibles podrían ser la dificultad de implementación del propio método, la dificultad de mejorar la optimización de código redundante entre funciones recursivas y el tratamiento de grandes cantidades de información, al expresar los algoritmos a tan bajo nivel.

Por un lado, la complejidad computacional de los algoritmos utilizados es muy baja, con cotas superiores asintóticas lineales. Por otro, la optimización parece muy buena, superando a una herramienta destinada a optimizar circuitos. Además, este método debería de escalar muy bien, ya que el fundamento es eliminar redundancia a bajo nivel, con lo cual cuanto más grandes sean los algoritmos utilizados, más posibilidades de optimizar redundancia.

Otra virtud destacable de este método es la paralelización. Dado que se representa cualquier algoritmo mediante funciones NAND, sabemos exactamente el coste de ejecución de cualquiera de sus partes. Imaginemos que tuviéramos acceso a un procesador que solo fuera capaz de ejecutar una única operación lógica NAND. Sería un procesador increíblemente sencillo y podríamos poner a trabajar en paralelo un número arbitrario de procesadores de ese tipo, aprovechando al máximo la paralelización obtenida a este nivel.

5.2 Propuestas de desarrollo futuro

Para que este proyecto pudiera convertirse en una herramienta comercial, haría falta, en primer lugar, aplicarlo a algoritmos arbitrariamente complejos y, en segundo lugar, implementar la traducción de la expresión optimizada en funciones NAND al lenguaje deseado.

5.2.1 Aplicación a algoritmos más complejos

Durante este proyecto se pretendía demostrar un concepto, por lo que el algoritmo más complejo que se ha definido ha sido el de la multiplicación (ver anexo 7.5). Se han sentado las bases, sin embargo, para poder implementar algoritmos de complejidad arbitraria. Una muestra sería, por ejemplo, el algoritmo de ordenación por burbuja.

Sería especialmente interesante aplicar este proyecto para optimizarse a sí mismo.

5.2.2 Traducción de vuelta del lenguaje intermedio optimizado al lenguaje original

Traducir el código de vuelta al lenguaje original no tiene el mismo interés teórico que optimizar los algoritmos, pero tiene un interés práctico.

El algoritmo de traducción aprovecharía la estructura recursiva de la base de datos de definiciones. Cuando se hallan todas las instancias de cada definición, se reemplazan por esta, de la más pequeña a la más grande.

5.2.3 Demostración de que el modelo de representación de algoritmos es Turing completo.

A lo largo de este proyecto, se ha asumido que el modelo utilizado para representar algoritmos a través de funciones recursivas es Turing completo. Parece bastante evidente, pero no se ha demostrado formalmente. Algunas posibilidades para hacerlo serían implementar una máquina de Turing, implementar el juego de la vida de Conway o definir todas las funciones μ-recursivas.

5.2.4 Implementación de un método de evaluación en anchura

Se ha realizado un prototipo de otro método de evaluación. A la hora de evaluar funciones recursivas con expresiones condicionales, es necesario un método de evaluación más inteligente, para evitar bucles infinitos.

Algunos de los requisitos son que la evaluación sea perezosa, en anchura y bit a bit.

5.2.5 Generalización de la optimización de algoritmos recursivos

Hasta ahora, hemos implementado un método de optimización para cuando una función es recursiva. Sin embargo no se ha implementado un método específico para cuando por ejemplo una función recursiva contiene otras funciones recursivas. Una posibilidad sería juntar todas las instancias a funciones recursivas en una única función recursiva.

5.2.6 Mejoras en la optimización

Una propuesta interesante sería mejorar aún más la optimización de código. Para ello podría, por ejemplo, generalizarse la técnica del sumador con generación anticipada de acarreo, calculando más rápidamente operaciones para múltiples entradas posibles.

También podríamos desenrollar llamadas recursivas, para minimizar la profundidad de las funciones.

Otra posibilidad sería analizar en mayor profundidad la recursión para que en cada llamada se realice el mínimo de operaciones necesarias para calcular un bit de salida.

Además, se podría mejorar la optimización con definiciones de funciones conocidas que son especialmente buenas, como ocurre con la función XOR.

6 Bibliografía

[1] A. W. Appel, de *Modern Compiler Implementation in ML*, Cambridge University Press, 1998, p. 401.

[2] R. S. J. U. A. Aho, Compiladores: principios, técnicas y herramientas., Addison-Wesley Iberoamericana, 1993.

[3] D. Buchfuhrer y C. Umans, «The complexity of Boolean formula minimization,» *Journal of Computer and System Sciences,* pp. 142-153, Enero 2011.

[4] W.-K. Chen, «The VLSI Handbook,» CRC Press, 2000.

[5] W. Wolf, Modern VLSI design systems on silicon, Prentice Hall, 1998.

[6] J. E.J. McCluskey, «Minimization of Boolean Functions,» *Bell System Technical Journal,* vol. 35, nº 6, p. 1417–1444, November 1956.

[7] G. H. C. M. A. S.-V. R.K. Brayton, Logic Minimization Algorithms for VLSI Synthesis, Norwell, MA: Kluwer Academic Publisher, 1984.

[8] R. R. a. A. Sangiovanni-Vincentelli., «Multiple-valued minmization for PLA optimization,» *IEEE Transactions on Computer-Aided Design,* vol. 6, nº 5, pp. 727-750, September 1987.

[9] 14 12 2016. [En línea]. Available: http://ethw.org/Robert_K._Brayton.

[10] «berkeley.edu Berkeley, University of California,» The Donald O. Pederson Center for Electronic Systems Design https://embedded.eecs.berkeley.edu/, [En línea]. Available: https://embedded.eecs.berkeley.edu/pubs/downloads/octtools/index.htm. [Último acceso: 16 2 2017].

[11] L. L. G. M. Louis Scheffer, «EDA for IC Implementation, Circuit Design, and Process Technology,» de *Taylor & Francis*, 2006, p. 42.

[12] C. M. G. D. H. y. A. S. R. K. Brayton, «Multilevel Logic Synthesis,» *Proceedings of the IEEE,* pp. 264-300, Febrero 1990.

[13] M. M. y. C. R. K. Mano, «Logic and Computer Design Fundamentals, Third Edition,» Prentice Hall, 2004, p. 73.

[14] Logic Friday, 11 11 2016. [En línea]. Available: http://www.sontrak.com/.

[15] Columbia University, «Columbia University,» 11 11 2016. [En línea]. Available: http://www.cs.columbia.edu/~cs6861/handouts/hw1-espresso-problem.txt.

[16] R. R. A. S.-V. A. W. RK Brayton, «MIS: A multiple-level logic optimization system,» *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* pp. 1062-1081, Noviembre 1987.

[17] C. E. L. R. L. R. a. C. S. Thomas H. Cormen, Introduction to Algorithms, Second Edition, MIT Press and McGraw–Hill, 2001.

7 Anexos

7.1 Desarrollo del proyecto

Al ser un proyecto de investigación, ha resultado imposible prever el coste real de desarrollo.

7.1.1 Implementación

El proceso de implementación se ha demorado durante algo más de un año por su complejidad.

Se ha seguido un proceso de refinamiento iterativo, tanto de los algoritmos, como sobre todo, del modelo de representación y las estructuras de datos usadas.

Un ejemplo de esta simplificación iterativa es que, en un principio, se utilizaban identificadores únicos para los nodos del bosque de funciones NAND. Afortunadamente, se vio que se podía realizar la misma función de forma más sencilla mediante una tabla dispersa bidimensional. Con ello se evitó usar una estructura de datos de crecimiento exponencial.

7.1.2 Imprevistos

Lo qué más tiempo ha llevado del proyecto ha sido simplificar el modelo usado para representar las funciones recursivas, lo que resulta curioso cuando el objetivo del proyecto es precisamente simplificar código. Esto ayuda a justificar el proyecto, siendo un ejemplo de lo tedioso que es el proceso de simplificación.

Muchos de los errores que se han producido posiblemente se hubieran podido evitar teniendo acceso a un experto en optimización de código y minimización de circuitos. Lamentablemente son dos campos que no hemos encontrado ligados en ninguna publicación.

Como ya se ha mencionado, el proceso de refinamiento del proyecto ha sido iterativo.

En una primera iteración se intentó desarrollar una estructura con nodos padres e hijos, sin embargo algunas de las operaciones resultaban imposibles con ese modelo.

En una segunda iteración se intentaron usar subnodos y supernodos, pero esto tampoco resultaba suficiente.

En una tercera iteración se usaban subnodos padre, supernodos padre, subnodos hijos y supernodos hijos. Esta estructura finalmente era adecuada para realizar todas las operaciones, pero muy compleja. En esta iteración se usaba un modelo de tres índices posibles para los subnodos: “primero”, “ultimo” y “resto”. Con esta iteración se llegó a optimizar completamente un algoritmo de suma, pero la implementación era demasiado complicada como para depurar operaciones más difíciles. Esta versión se encuentra congelada en una rama del repositorio público.[2]

Con cada una de estas revisiones ha sido necesario reescribir la mayor parte del código. La situación es intrínseca a una investigación primaria. Por la misma razón ha sido necesario emplear el modelo de desarrollo incremental. Se han necesitado varias iteraciones de soluciones aproximadas al problema, hasta dar con una satisfactoria. Durante dichas iteraciones, aparecen errores imprevistos que hacen cambiar el diseño, en ocasiones partes fundamentales de este.

[2] https://github.com/raescartin/Recompiler/tree/fission

El peso de la depuración de código ha sido mucho mayor del estimado, suponiendo una gran parte del tiempo de desarrollo.

La naturaleza del proyecto ha hecho que hubiera que modificar otros aspectos del diseño original. Por ejemplo, en algunas estructuras, se ha eliminado la copia de datos, sobrescribiendo los originales en su lugar, para hacer más fácil la depuración del código.

7.2 Métricas

Para el proyecto se han escrito más de 23,854 líneas de código y eliminado 18,053, tal y como se puede ver en el repositorio principal público en GitHub.[3]

Esta medida da una idea de lo complejo e iterativo del proceso de implementación del proyecto.

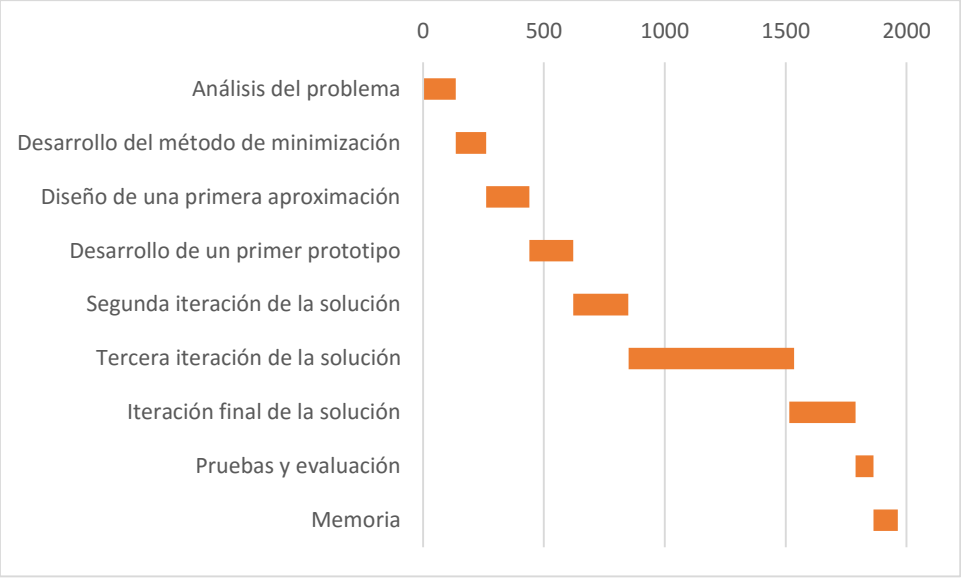


Figura 23: Diagrama de Gantt de horas dedicadas a las fases del proyecto

Este diagrama de Gantt nos muestra el número de horas dedicadas a las diferentes fases del proyecto. Hay que destacar la cantidad de horas empleadas con la tercera iteración. Esto se debe a que la complejidad del modelo hacía muy difícil depurar el código.

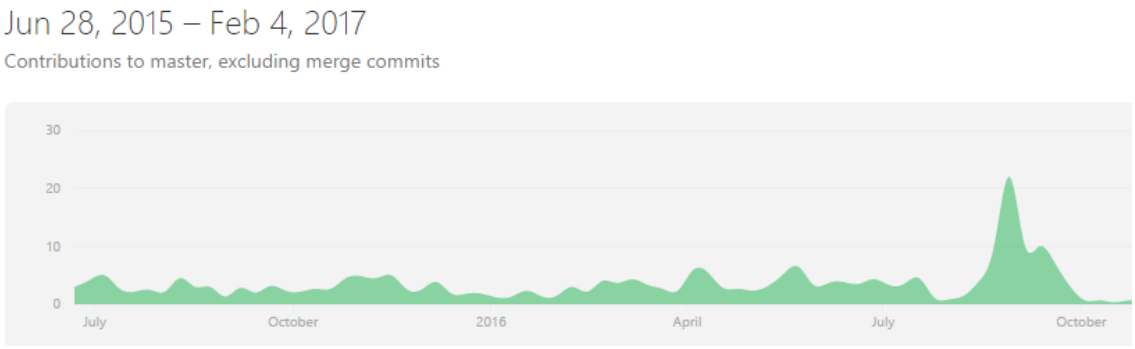


Figura 24: Contribuciones al repositorio principal

Tal y como se puede comprobar en el repositorio, la primera contribución de código se produce el 28 de junio de 2015. Desde entonces las contribuciones siguen un ritmo constante, con una

[3] https://github.com/raescartin/Recompiler/graphs/contributors

aceleración al final, debida más a una actualización frecuente del código que a un aumento de la producción.

El valle que se observa antes de esta aceleración corresponde al momento en que el proyecto ya era completamente funcional.

En total, se han dedicado unas 1964 horas a lo largo de casi dos años.

7.3 Sugerencias para la generalización de la optimización de funciones recursivas

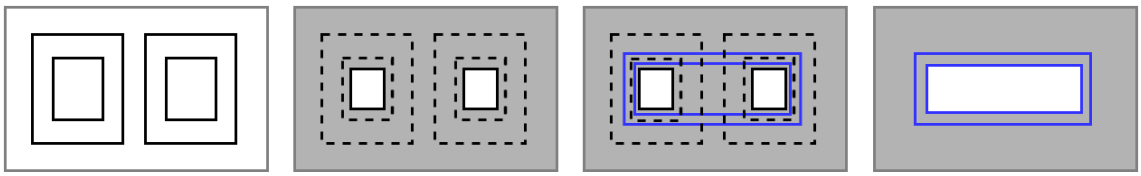


Figura 25: Optimización de funciones con múltiples instancias de funciones recursivas

A la hora de optimizar algoritmos que contengan múltiples funciones recursivas, podría utilizarse un proceso similar al usado en la sección 3.7, pero juntando todos los nodos de entrada a funciones recursivas y todos los nodos de salida. De esta forma fusionaríamos todas las funciones recursivas en una sola.

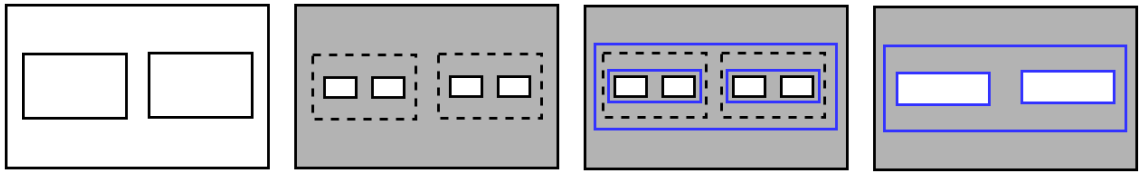


Figura 26: Optimización de un algoritmo con varias instancias recursivas

Del mismo modo podríamos optimizar un algoritmo que tenga múltiples llamadas recursivas a sí mismo.

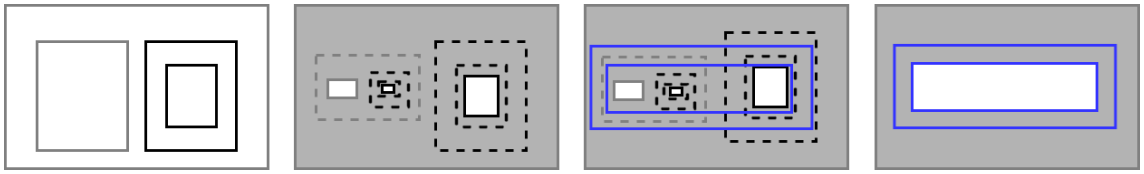


Figura 27: Algoritmo recursivo con funciones recursivas

Finalmente, podemos mezclar los dos casos anteriores a fin de optimizar algoritmos recursivos que contengan otras funciones recursivas.

7.4 Tabla de verdad de la suma de 4 bits

A3	B3	A2	B2	A1	B1	A0	B0	S4	S3	S2	S1	S0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	1	1	0	0	0	1	0
0	0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	0	1	0	1	0	0	0	1	1
0	0	0	0	0	1	1	0	0	0	0	1	1
0	0	0	0	0	1	1	1	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0
0	0	0	0	1	0	0	1	0	0	0	1	1
0	0	0	0	1	0	1	0	0	0	0	1	1
0	0	0	0	1	0	1	1	0	0	1	0	0
0	0	0	0	1	1	0	0	0	0	1	0	0
0	0	0	0	1	1	0	1	0	0	1	0	1
0	0	0	0	1	1	1	0	0	0	1	0	1
0	0	0	0	1	1	1	1	0	0	1	1	0
0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	1	0	0	0	1	0	0	1	0	1
0	0	0	1	0	0	1	0	0	0	1	0	1
0	0	0	1	0	0	1	1	0	0	1	1	0
0	0	0	1	0	1	0	0	0	0	1	1	0
0	0	0	1	0	1	0	1	0	0	1	1	1
0	0	0	1	0	1	1	0	0	0	1	1	1
0	0	0	1	0	1	1	1	0	1	0	0	0
0	0	0	1	1	0	0	0	0	0	1	1	0
0	0	0	1	1	0	0	1	0	0	1	1	1
0	0	0	1	1	0	1	0	0	0	1	1	1
0	0	0	1	1	0	1	1	0	1	0	0	0
0	0	0	1	1	1	0	0	0	1	0	0	0
0	0	0	1	1	1	0	1	0	1	0	0	1
0	0	0	1	1	1	1	0	0	1	0	0	1
0	0	0	1	1	1	1	1	0	1	0	1	0
0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	1	0	0	1	0	1
0	0	1	0	0	0	1	0	0	0	1	0	1
0	0	1	0	0	0	1	1	0	0	1	1	0
0	0	1	0	0	1	0	0	0	0	1	1	0
0	0	1	0	0	1	0	1	0	0	1	1	1
0	0	1	0	0	1	1	0	0	0	1	1	1
0	0	1	0	0	1	1	1	0	1	0	0	0
0	0	1	0	1	0	0	0	0	0	1	1	0
0	0	1	0	1	0	0	1	0	0	1	1	1
0	0	1	0	1	0	1	0	0	0	1	1	1
0	0	1	0	1	0	1	1	0	1	0	0	0
0	0	1	0	1	1	0	0	0	1	0	0	0
0	0	1	0	1	1	0	1	0	1	0	0	1
0	0	1	0	1	1	1	0	0	1	0	0	1
0	0	1	0	1	1	1	1	0	1	0	1	0
0	0	1	1	0	0	0	0	0	1	0	0	0
0	0	1	1	0	0	0	1	0	1	0	0	1
0	0	1	1	0	0	1	0	0	1	0	0	1
0	0	1	1	0	0	1	1	0	1	0	1	0
0	0	1	1	0	1	0	0	0	1	0	1	0
0	0	1	1	0	1	0	1	0	1	0	1	1
0	0	1	1	0	1	1	0	0	1	0	1	1
0	0	1	1	0	1	1	1	0	1	1	0	0
0	0	1	1	1	0	0	0	0	1	0	1	0
0	0	1	1	1	0	0	1	0	1	0	1	1
0	0	1	1	1	0	1	0	0	1	0	1	1
0	0	1	1	1	0	1	1	0	1	1	0	0
0	0	1	1	1	1	0	0	0	1	1	0	0
0	0	1	1	1	1	0	1	0	1	1	0	1
0	0	1	1	1	1	1	0	0	1	1	0	1
0	0	1	1	1	1	1	1	0	1	1	1	0
0	1	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	1	0	1	0	0	1

0	1	0	0	0	0	1	0		0	1	0	0	1
0	1	0	0	0	0	1	1		0	1	0	1	0
0	1	0	0	0	1	0	0		0	1	0	1	0
0	1	0	0	0	1	0	1		0	1	0	1	1
0	1	0	0	0	1	1	0		0	1	0	1	1
0	1	0	0	0	1	1	1		0	1	1	0	0
0	1	0	0	1	0	0	0		0	1	0	1	0
0	1	0	0	1	0	0	1		0	1	0	1	1
0	1	0	0	1	0	1	0		0	1	0	1	1
0	1	0	0	1	0	1	1		0	1	1	0	0
0	1	0	0	1	1	0	0		0	1	1	0	0
0	1	0	0	1	1	0	1		0	1	1	0	1
0	1	0	0	1	1	1	0		0	1	1	0	1
0	1	0	0	1	1	1	1		0	1	1	1	0
0	1	0	1	0	0	0	0		0	1	1	0	0
0	1	0	1	0	0	0	1		0	1	1	0	1
0	1	0	1	0	0	1	0		0	1	1	0	1
0	1	0	1	0	0	1	1		0	1	1	1	0
0	1	0	1	0	1	0	0		0	1	1	1	0
0	1	0	1	0	1	0	1		0	1	1	1	1
0	1	0	1	0	1	1	0		0	1	1	1	1
0	1	0	1	0	1	1	1		1	0	0	0	0
0	1	0	1	1	0	0	0		0	1	1	1	0
0	1	0	1	1	0	0	1		0	1	1	1	1
0	1	0	1	1	0	1	0		0	1	1	1	1
0	1	0	1	1	0	1	1		1	0	0	0	0
0	1	0	1	1	1	0	0		1	0	0	0	0
0	1	0	1	1	1	0	1		1	0	0	0	1
0	1	0	1	1	1	1	0		1	0	0	0	1
0	1	0	1	1	1	1	1		1	0	0	1	0
0	1	1	0	0	0	0	0		0	1	1	0	0
0	1	1	0	0	0	0	1		0	1	1	0	1
0	1	1	0	0	0	1	0		0	1	1	0	1
0	1	1	0	0	0	1	1		0	1	1	1	0
0	1	1	0	0	1	0	0		0	1	1	1	0
0	1	1	0	0	1	0	1		0	1	1	1	1
0	1	1	0	0	1	1	0		0	1	1	1	1
0	1	1	0	0	1	1	1		1	0	0	0	0
0	1	1	0	1	0	0	0		0	1	1	1	0
0	1	1	0	1	0	0	1		0	1	1	1	1
0	1	1	0	1	0	1	0		0	1	1	1	1
0	1	1	0	1	0	1	1		1	0	0	0	0
0	1	1	0	1	1	0	0		1	0	0	0	0
0	1	1	0	1	1	0	1		1	0	0	0	1
0	1	1	0	1	1	1	0		1	0	0	0	1
0	1	1	0	1	1	1	1		1	0	0	1	0
0	1	1	1	0	0	0	0		1	0	0	0	0
0	1	1	1	0	0	0	1		1	0	0	0	1
0	1	1	1	0	0	1	0		1	0	0	0	1
0	1	1	1	0	0	1	1		1	0	0	1	0
0	1	1	1	0	1	0	0		1	0	0	1	0
0	1	1	1	0	1	0	1		1	0	0	1	1
0	1	1	1	0	1	1	0		1	0	0	1	1
0	1	1	1	0	1	1	1		1	0	1	0	0
0	1	1	1	1	0	0	0		1	0	0	1	0
0	1	1	1	1	0	0	1		1	0	0	1	1
0	1	1	1	1	0	1	0		1	0	0	1	1
0	1	1	1	1	0	1	1		1	0	1	0	0
0	1	1	1	1	1	0	0		1	0	1	0	0
0	1	1	1	1	1	0	1		1	0	1	0	1
0	1	1	1	1	1	1	0		1	0	1	0	1
0	1	1	1	1	1	1	1		1	0	1	1	0
1	0	0	0	0	0	0	0		0	1	0	0	0
1	0	0	0	0	0	0	1		0	1	0	0	1
1	0	0	0	0	0	1	0		0	1	0	0	1
1	0	0	0	0	0	1	1		0	1	0	1	0
1	0	0	0	0	1	0	0		0	1	0	1	0
1	0	0	0	0	1	0	1		0	1	0	1	1
1	0	0	0	0	1	1	0		0	1	0	1	1
1	0	0	0	0	1	1	1		0	1	1	0	0

1	0	0	0	1	0	0	0	0	1	0	1	0
1	0	0	0	1	0	0	1	0	1	0	1	1
1	0	0	0	1	0	1	0	0	1	0	1	1
1	0	0	0	1	0	1	1	0	1	1	0	0
1	0	0	0	1	1	0	0	0	1	1	0	0
1	0	0	0	1	1	0	1	0	1	1	0	1
1	0	0	0	1	1	1	0	0	1	1	0	1
1	0	0	0	1	1	1	1	0	1	1	1	0
1	0	0	1	0	0	0	0	0	1	1	0	0
1	0	0	1	0	0	0	1	0	1	1	0	1
1	0	0	1	0	0	1	0	0	1	1	0	1
1	0	0	1	0	0	1	1	0	1	1	1	0
1	0	0	1	0	1	0	0	0	1	1	1	0
1	0	0	1	0	1	0	1	0	1	1	1	1
1	0	0	1	0	1	1	0	0	1	1	1	1
1	0	0	1	0	1	1	1	1	0	0	0	0
1	0	0	1	1	0	0	0	0	1	1	1	0
1	0	0	1	1	0	0	1	0	1	1	1	1
1	0	0	1	1	0	1	0	0	1	1	1	1
1	0	0	1	1	0	1	1	1	0	0	0	0
1	0	0	1	1	1	0	0	0	1	0	0	0
1	0	0	1	1	1	0	0	1	1	0	0	0
1	0	0	1	1	1	0	1	0	1	0	0	1
1	0	0	1	1	1	0	1	1	1	0	0	1
1	0	0	1	1	1	1	0	0	1	0	0	0
1	0	0	1	1	1	1	0	1	1	0	0	0	1
1	0	0	1	1	1	1	1	0	1	0	0	0	1
1	0	0	1	1	1	1	1	1	1	0	0	1	0
1	0	1	0	0	0	0	0	0	0	1	1	0	0
1	0	1	0	0	0	0	0	1	0	1	1	0	1
1	0	1	0	0	0	0	1	0	0	1	1	0	1
1	0	1	0	0	0	0	1	1	0	1	1	1	0
1	0	1	0	0	0	1	0	0	0	1	1	1	0
1	0	1	0	0	0	1	0	1	0	1	1	1	1
1	0	1	0	0	0	1	1	0	0	1	1	1	1
1	0	1	0	0	0	1	1	1	1	0	0	0	0
1	0	1	0	1	0	0	0	0	0	1	1	1	0
1	0	1	0	1	0	0	0	1	0	1	1	1	1
1	0	1	0	1	0	1	0	0	0	1	1	1	1
1	0	1	0	1	0	1	1	1	1	0	0	0	0
1	0	1	0	1	1	0	0	0	1	0	0	0	0
1	0	1	0	1	1	0	0	1	1	0	0	0	1
1	0	1	0	1	1	1	0	0	1	0	0	0	1
1	0	1	0	1	1	1	1	0	1	0	0	1	0
1	0	1	1	0	0	0	0	0	1	0	0	0	0
1	0	1	1	0	0	0	0	1	1	0	0	0	1
1	0	1	1	0	0	1	0	0	1	0	0	0	1
1	0	1	1	0	0	1	1	0	1	0	0	1	0
1	0	1	1	0	1	0	0	0	1	0	0	1	0
1	0	1	1	0	1	0	1	0	1	0	0	1	1
1	0	1	1	0	1	1	0	0	1	0	0	1	1
1	0	1	1	0	1	1	1	0	1	0	1	0	0
1	0	1	1	1	0	0	0	0	1	0	0	1	0
1	0	1	1	1	0	0	0	1	1	0	0	1	1
1	0	1	1	1	0	1	0	0	1	0	0	1	1
1	0	1	1	1	0	1	1	0	1	0	1	0	0
1	0	1	1	1	1	0	0	0	1	0	1	0	0
1	0	1	1	1	1	0	0	1	1	0	1	0	1
1	0	1	1	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	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	1	0	0	0	1
1	1	0	0	0	0	0	1	0	1	0	0	0	1
1	1	0	0	0	0	0	1	1	1	0	0	1	0
1	1	0	0	0	0	1	0	0	1	0	0	1	0
1	1	0	0	0	0	1	0	1	1	0	0	1	1
1	1	0	0	0	0	1	1	0	1	0	0	1	1
1	1	0	0	0	0	1	1	1	1	0	1	0	0
1	1	0	0	1	0	0	0	0	1	0	0	1	0
1	1	0	0	1	0	0	0	1	1	0	0	1	1
1	1	0	0	1	0	1	0	0	1	0	0	1	1
1	1	0	0	1	0	1	1	1	1	0	1	0	0
1	1	0	0	1	1	0	0	0	1	0	1	0	0
1	1	0	0	1	1	0	0	1	1	0	1	0	1

1	1	0	0	1	1	1	0	1	0	1	0	1
1	1	0	0	1	1	1	1	1	0	1	1	0
1	1	0	1	0	0	0	0	1	0	1	0	0
1	1	0	1	0	0	0	1	1	0	1	0	1
1	1	0	1	0	0	1	0	1	0	1	0	1
1	1	0	1	0	0	1	1	1	0	1	1	0
1	1	0	1	0	1	0	0	1	0	1	1	0
1	1	0	1	0	1	0	1	1	0	1	1	1
1	1	0	1	0	1	1	0	1	0	1	1	1
1	1	0	1	0	1	1	1	1	1	0	0	0
1	1	0	1	1	0	0	0	1	0	1	1	0
1	1	0	1	1	0	0	1	1	0	1	1	1
1	1	0	1	1	0	1	0	1	0	1	1	1
1	1	0	1	1	0	1	1	1	1	0	0	0
1	1	0	1	1	1	0	0	1	1	0	0	0
1	1	0	1	1	1	0	1	1	1	0	0	1
1	1	0	1	1	1	1	0	1	1	0	0	1
1	1	0	1	1	1	1	1	1	1	0	1	0
1	1	1	0	0	0	0	0	1	0	1	0	0
1	1	1	0	0	0	0	1	1	0	1	0	1
1	1	1	0	0	0	1	0	1	0	1	0	1
1	1	1	0	0	0	1	1	1	0	1	1	0
1	1	1	0	0	1	0	0	1	0	1	1	0
1	1	1	0	0	1	0	1	1	0	1	1	1
1	1	1	0	0	1	1	0	1	0	1	1	1
1	1	1	0	0	1	1	1	1	1	0	0	0
1	1	1	0	1	0	0	0	1	0	1	1	0
1	1	1	0	1	0	0	1	1	0	1	1	1
1	1	1	0	1	0	1	0	1	0	1	1	1
1	1	1	0	1	0	1	1	1	1	0	0	0
1	1	1	0	1	1	0	0	1	1	0	0	0
1	1	1	0	1	1	0	1	1	1	0	0	1
1	1	1	0	1	1	1	0	1	1	0	0	1
1	1	1	0	1	1	1	1	1	1	0	1	0
1	1	1	1	0	0	0	0	1	1	0	0	0
1	1	1	1	0	0	0	1	1	1	0	0	1
1	1	1	1	0	0	1	0	1	1	0	0	1
1	1	1	1	0	0	1	1	1	1	0	1	0
1	1	1	1	0	1	0	0	1	1	0	1	0
1	1	1	1	0	1	0	1	1	1	0	1	1
1	1	1	1	0	1	1	0	1	1	0	1	1
1	1	1	1	0	1	1	1	1	1	1	0	0
1	1	1	1	1	0	0	0	1	1	0	1	0
1	1	1	1	1	0	0	1	1	1	0	1	1
1	1	1	1	1	0	1	0	1	1	0	1	1
1	1	1	1	1	0	1	1	1	1	1	0	0
1	1	1	1	1	1	0	0	1	1	1	0	0
1	1	1	1	1	1	0	1	1	1	1	0	1
1	1	1	1	1	1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	0

7.5 Algoritmo de multiplicación

Se ha implementado un algoritmo de multiplicación.

Este algoritmo usa la definición del algoritmo andRecursivo que aparece en la sección 3.1 y el algoritmo de la suma de la sección 4.3.

Su funcionamiento está inspirado en el algoritmo de la suma. Se calcula un bit de salida en cada iteración del algoritmo recursivo mulR.

mul[0,1,2(4{n}&5&3{n})] =

andRecursivo[1{n},0;3]

mulR[0,1{1..n-1},3{1..n-1};4,5]

mulR[0,1,2;3(7&6{1..n-1}),4(8&6{n})]=

andR[1{n},0;5]

add[2,5;6]

mulR[0,1{1..n-1},6{1..n-1};7,8]

7.6 Traza completa de la optimización de la suma modular

Durante la tercera iteración del proyecto se consiguió, por fin, optimizar un algoritmo recursivo, teniendo en cuenta la redundancia entre iteraciones.

A continuación se presenta una traza de la optimización completa de la suma modular. Es un algoritmo de suma dónde las entradas tienen la misma longitud *n* en bits y se calcula la salida para esa misma longitud *n* en bits.

Hay que tener que cuando se optimizó este algoritmo la notación era algo diferente. Se usaban tres índices para dividir un nodo en el primer elemento, último elemento y el resto. Los índices de estos elementos son {1}, {3} y {2}, respectivamente.

El algoritmo inicial para la suma modular es el siguiente:

add[0,1; 2(17&11{2})] =

xor[0,1; 11] and[9(0{1}&0{2}),10(1{1}&1{2}); 16]

add[15(11{0}&11{1}),16; 17]

7.6.1 Optimización de la parte secuencial

En primer lugar se optimiza la parte secuencial. Para ello se expresa esta como funciones NAND.

add[0,1,17; 2(17&11{2}),15(11{0}&11{1}),16] =

nand [0,0; 18] nand [1,1; 19] nand [0,1; 21] nand [9(0{1}&0{2}),10(1{1}&1{2}); 23]

nand [18,19; 20] nand [23,23; 16]

nand [20,21; 22]

nand [22,22; 11]

A continuación se realiza la fisión de nodos.

add[0,1,17; 2(17&11{2}),15(11{0}&11{1}),16(29{1}&29{2})] =

nand [0,0; 18] nand [1,1; 19] nand [0,1; 21] nand [9(0{1}&0{2}),10(1{1}&1{2}); 23(25{1}&25{2})] nand [0{0},0{0}; 18{0}] nand [0{1},0{1}; 18{1}] nand [0{2},0{2}; 18{2}] nand [1{0},1{0}; 19{0}] nand [1{1},1{1}; 19{1}] nand [1{2},1{2}; 19{2}] nand [0{0},1{0}; 21{0}] nand [0{1},1{1}; 21{1}] nand [0{2},1{2}; 21{2}] nand [0{1},1{1}; 25{1}] nand [0{2},1{2}; 25{2}]

nand [18,19; 20] nand [23(25{1}&25{2}),23(25{1}&25{2}); 16(29{1}&29{2})] nand [18{0},19{0}; 20{0}] nand [18{1},19{1}; 20{1}] nand [18{2},19{2}; 20{2}] nand [25{1},25{1}; 29{1}] nand [25{2},25{2}; 29{2}]

nand [20,21; 22] nand [20{0},21{0}; 22{0}] nand [20{1},21{1}; 22{1}] nand [20{2},21{2}; 22{2}]

nand [22,22; 11] nand [22{0},22{0}; 11{0}] nand [22{1},22{1}; 11{1}] nand [22{2},22{2}; 11{2}]

Se obtiene el bosque de funciones NAND y a partir de él los nodos equivalentes.

equivalentNodes:{25{1}=21{1}, 25{2}=21{2}}

Se reemplazan los nodos equivalentes en la definición original del algoritmo.

add[0,1,17; 2(17&11{2}),15(11{0}&11{1}),16(29{1}&29{2})] =

nand [0,0; 18] nand [1,1; 19] nand [0,1; 21] nand [9(0{1}&0{2}),10(1{1}&1{2}); 23(21{1}&21{2})] nand [0{0},0{0}; 18{0}] nand [0{1},0{1}; 18{1}] nand [0{2},0{2}; 18{2}] nand [1{0},1{0}; 19{0}] nand [1{1},1{1}; 19{1}] nand [1{2},1{2}; 19{2}] nand [0{0},1{0}; 21{0}] nand [0{1},1{1}; 21{1}] nand [0{2},1{2}; 21{2}]

nand [18,19; 20] nand [23(21{1}&21{2}),23(21{1}&21{2}); 16(29{1}&29{2})] nand [18{0},19{0}; 20{0}] nand [18{1},19{1}; 20{1}] nand [18{2},19{2}; 20{2}] nand [21{1},21{1}; 29{1}] nand [21{2},21{2}; 29{2}]

nand [20,21; 22] nand [20{0},21{0}; 22{0}] nand [20{1},21{1}; 22{1}] nand [20{2},21{2}; 22{2}]

nand [22,22; 11] nand [22{0},22{0}; 11{0}] nand [22{1},22{1}; 11{1}] nand [22{2},22{2}; 11{2}]

Se realiza la fusion de nodos.

add[0,1,17; 2(17&11{2}),15(11{0}&11{1}),16] =

nand [0,0; 18] nand [1,1; 19] nand [0,1; 21]

nand [18,19; 20] nand [23(21{1}&21{2}),23(21{1}&21{2}); 16]

nand [20,21; 22]

nand [22,22; 11]

Y se reconstruye la función.

add[0,1,2; 29(2&24{2}),26(24{0}&24{1}),30] =

nand [0,0; 3] nand [1,1; 7] nand [0,1; 15]

nand [3,7; 11] nand [18(15{1}&15{2}),18(15{1}&15{2}); 30]

nand [11,15; 20]

nand [20,20; 24]

Después se recupera la recursividad.

add[0,1; 29(2&24{2})] =

nand [0,0; 3] nand [1,1; 7] nand [0,1; 15]

nand [3,7; 11] nand [18(15{1}&15{2}),18(15{1}&15{2}); 30]

nand [11,15; 20]

nand [20,20; 24]

add [26(24{0}&24{1}),30; 2]

7.6.2 Optimización de la redundancia entre iteraciones

Partimos de la función:

addCopy[0,1; 18(17&7{2})] =

nand [0,0; 2] nand [1,1; 3] nand [0,1; 5]

nand [2,3; 4] nand [15(5{1}&5{2}),15(5{1}&5{2}); 16]

nand [4,5; 6]

nand [6,6; 7]

add [11(7{0}&7{1}),16; 17]

En primer lugar se expande la recursión, asociando los nodos expandidos a los nodos desde los que se expanden en una tabla.

addCopy[0,1,28; 18(17(28&30{2})&7{2}),55(30{0}&30{1}),50] =

nand [0,0; 2] nand [1,1; 3] nand [0,1; 5]

nand [2,3; 4] nand [15(5{1}&5{2}),15(5{1}&5{2}); 16]

nand [4,5; 6] nand [16,16; 37]

nand [6,6; 7]

nand [11(7{0}&7{1}),11(7{0}&7{1}); 33] nand [11(7{0}&7{1}),16; 41]

nand [33,37; 45] nand [49(41{1}&41{2}),49(41{1}&41{2}); 50]

nand [45,41; 51]

nand [51,51; 30]

A continuación se realiza la fisión de los nodos.

addCopy[0,1,28;
18(17(28&30(81{0}&81{1}){2}(81{1}{2}))&7{2}),55(30(81{0}&81{1}){0}(81{0})&30(81{0}&81{1}){1}(81{1}{0}&81{1}{1})),50(91{1}(95{0}&95{1})&91{2})] =

nand [0,0; 2] nand [1,1; 3] nand [0,1; 5] nand [0{0},0{0}; 2{0}] nand [0{1},0{1}; 2{1}] nand [0{2},0{2}; 2{2}] nand [1{0},1{0}; 3{0}] nand [1{1},1{1}; 3{1}] nand [1{2},1{2}; 3{2}] nand [0{0},1{0}; 5{0}] nand [0{1},1{1}; 5{1}] nand [0{2},1{2}; 5{2}] nand [0{1}{0},0{1}{0}; 2{1}{0}] nand [0{1}{1},0{1}{1}; 2{1}{1}] nand [0{1}{2},0{1}{2}; 2{1}{2}] nand [1{1}{0},1{1}{0}; 3{1}{0}] nand [1{1}{1},1{1}{1}; 3{1}{1}] nand [1{1}{2},1{1}{2}; 3{1}{2}] nand [0{1}{0},1{1}{0}; 5{1}{0}] nand [0{1}{1},1{1}{1}; 5{1}{1}] nand [0{1}{2},1{1}{2}; 5{1}{2}]

nand [2,3; 4] nand [15(5{1}&5{2}),15(5{1}&5{2}); 16(61{1}&61{2})] nand [2{0},3{0}; 4{0}] nand [2{1},3{1}; 4{1}] nand [2{2},3{2}; 4{2}] nand [2{1}{0},3{1}{0}; 4{1}{0}] nand [2{1}{1},3{1}{1}; 4{1}{1}] nand [2{1}{2},3{1}{2}; 4{1}{2}] nand [15(5{1}&5{2}){0}(5{1}{0}),15(5{1}&5{2}){0}(5{1}{0}); 61{1}{0}] nand [15(5{1}&5{2}){1}(5{1}{1}&5{1}{2}),15(5{1}&5{2}){1}(5{1}{1}&5{1}{2}); 16(61{1}&61{2}){1}(61{1}{1}&61{1}{2})] nand [15(5{1}&5{2}){2}(5{2}),15(5{1}&5{2}){2}(5{2}); 61{2}] nand [5{1}{1},5{1}{1}; 61{1}{1}] nand [5{1}{2},5{1}{2}; 61{1}{2}]

nand [4,5; 6] nand [16(61{1}&61{2}),16(61{1}&61{2}); 37(65{1}&65{2})] nand [4{0},5{0}; 6{0}] nand [4{1},5{1}; 6{1}] nand [4{2},5{2}; 6{2}] nand [4{1}{0},5{1}{0}; 6{1}{0}] nand [4{1}{1},5{1}{1}; 6{1}{1}] nand [4{1}{2},5{1}{2}; 6{1}{2}] nand [16(61{1}&61{2}){0}(61{1}{0}),16(61{1}&61{2}){0}(61{1}{0}); 65{1}{0}] nand [16(61{1}&61{2}){1}(61{1}{1}&61{1}{2}),16(61{1}&61{2}){1}(61{1}{1}&61{1}{2}); 37(65{1}&65{2}){1}(65{1}{1}&65{1}{2})] nand [16(61{1}&61{2}){2}(61{2}),16(61{1}&61{2}){2}(61{2}); 65{2}] nand [61{1}{1},61{1}{1}; 65{1}{1}] nand [61{1}{2},61{1}{2}; 65{1}{2}]

nand [6,6; 7] nand [6{0},6{0}; 7{0}] nand [6{1},6{1}; 7{1}] nand [6{2},6{2}; 7{2}] nand [6{1}{0},6{1}{0}; 7{1}{0}] nand [6{1}{1},6{1}{1}; 7{1}{1}] nand [6{1}{2},6{1}{2}; 7{1}{2}]

nand [11(7{0}&7{1}),11(7{0}&7{1}); 33(57{0}&57{1})] nand [11(7{0}&7{1}),16(61{1}&61{2}); 41(73{0}&73{1})] nand [11(7{0}&7{1}){0}(7{0}),11(7{0}&7{1}){0}(7{0}); 57{0}] nand [11(7{0}&7{1}){1}(7{1}{0}&7{1}{1}),11(7{0}&7{1}){1}(7{1}{0}&7{1}{1}); 33(57{0}&57{1}){1}(57{1}{0}&57{1}{1})] nand

[11(7{0}&7{1}){2}(7{1}{2}),11(7{0}&7{1}){2}(7{1}{2}); 57{1}{2}] nand [11(7{0}&7{1}){0}(7{0}),16(61{1}&61{2}){0}(61{1}{0}); 73{0}] nand [11(7{0}&7{1}){1}(7{1}{0}&7{1}{1}),16(61{1}&61{2}){1}(61{1}{1}&61{1}{2}); 41(73{0}&73{1}){1}(73{1}{0}&73{1}{1})] nand [11(7{0}&7{1}){2}(7{1}{2}),16(61{1}&61{2}){2}(61{2}); 73{1}{2}] nand [7{1}{0},7{1}{0}; 57{1}{0}] nand [7{1}{1},7{1}{1}; 57{1}{1}] nand [7{1}{0},61{1}{1}; 73{1}{0}] nand [7{1}{1},61{1}{2}; 73{1}{1}]

nand [33(57{0}&57{1}),37(65{1}&65{2}); 45(69{0}&69{1})] nand [49(41(73{0}&73{1}){1}(73{1}{0}&73{1}{1})&41(73{0}&73{1}){2}(73{1}{2})),49(41(73{0}&73{1}){1}(73{1}{0}&73{1}{1})&41(73{0}&73{1}){2}(73{1}{2})); 50(91{1}(95{0}&95{1})&91{2})] nand [33(57{0}&57{1}){0}(57{0}),37(65{1}&65{2}){0}(65{1}{0}); 69{0}] nand [33(57{0}&57{1}){1}(57{1}{0}&57{1}{1}),37(65{1}&65{2}){1}(65{1}{1}&65{1}{2}); 45(69{0}&69{1}){1}(69{1}{0}&69{1}{1})] nand [33(57{0}&57{1}){2}(57{1}{2}),37(65{1}&65{2}){2}(65{2}); 69{1}{2}] nand [57{1}{0},65{1}{1}; 69{1}{0}] nand [57{1}{1},65{1}{2}; 69{1}{1}] nand [41(73{0}&73{1}){1}(73{1}{0}&73{1}{1}),41(73{0}&73{1}){1}(73{1}{0}&73{1}{1}); 91{1}(95{0}&95{1})] nand [41(73{0}&73{1}){2}(73{1}{2}),41(73{0}&73{1}){2}(73{1}{2}); 91{2}] nand [73{1}{0},73{1}{0}; 95{0}] nand [73{1}{1},73{1}{1}; 95{1}]

nand [45(69{0}&69{1}),41(73{0}&73{1}); 51(77{0}&77{1})] nand [45(69{0}&69{1}){0}(69{0}),41(73{0}&73{1}){0}(73{0}); 77{0}] nand [45(69{0}&69{1}){1}(69{1}{0}&69{1}{1}),41(73{0}&73{1}){1}(73{1}{0}&73{1}{1}); 51(77{0}&77{1}){1}(77{1}{0}&77{1}{1})] nand [45(69{0}&69{1}){2}(69{1}{2}),41(73{0}&73{1}){2}(73{1}{2}); 77{1}{2}] nand [69{1}{0},73{1}{0}; 77{1}{0}] nand [69{1}{1},73{1}{1}; 77{1}{1}]

nand [51(77{0}&77{1}),51(77{0}&77{1}); 30(81{0}&81{1})] nand [51(77{0}&77{1}){0}(77{0}),51(77{0}&77{1}){0}(77{0}); 81{0}] nand [51(77{0}&77{1}){1}(77{1}{0}&77{1}{1}),51(77{0}&77{1}){1}(77{1}{0}&77{1}{1}); 30(81{0}&81{1}){1}(81{1}{0}&81{1}{1})] nand [51(77{0}&77{1}){2}(77{1}{2}),51(77{0}&77{1}){2}(77{1}{2}); 81{1}{2}] nand [77{1}{0},77{1}{0}; 81{1}{0}] nand [77{1}{1},77{1}{1}; 81{1}{1}]

Después se transforma a un bosque de funciones NAND y obtenemos los nodos equivalentes.

equivalentNodes:{65{1}{1}=5{1}{1}, 57{1}{1}=6{1}{1}, 65{1}{2}=5{1}{2}, 65{2}=5{2}, 65{1}{0}=5{1}{0}, 57{1}{2}=6{1}{2}, 57{0}=6{0}, 57{1}{0}=6{1}{0}}

equivalentNodes:{57{1}{1}=6{1}{1}, 65{1}{2}=5{1}{2}, 37(65{1}&65{2}){2}(65{2})=15(5{1}&5{2}){2}(5{2}), 65{2}=5{2}, 65{1}=5{1}, 57{1}=6{1}, 57{0}=6{0}, 65{1}{1}=5{1}{1}, 37(65{1}&65{2}){1}(65{1}{1}&65{1}{2})=15(5{1}&5{2}){1}(5{1}{1}&5{1}{2}), 65{1}{0}=5{1}{0}, 57{1}{2}=6{1}{2}, 37(65{1}&65{2})=15(5{1}&5{2}), 57{1}{0}=6{1}{0}, 37(65{1}&65{2}){0}(65{1}{0})=15(5{1}&5{2}){0}(5{1}{0})}

Se reemplazan los nodos equivalentes de la definición original.

addCopy[0,1,28;
18(17(28&30(81{0}&81{1}){2}(81{1}{2}))&7{2}),55(30(81{0}&81{1}){0}(81{0})&30(81{0}&81{1}){1}(81{1}{0}&81{1}{1})),50(91{1}(95{0}&95{1})&91{2})] =

nand [0,0; 2] nand [1,1; 3] nand [0,1; 5] nand [0{0},0{0}; 2{0}] nand [0{1},0{1}; 2{1}] nand [0{2},0{2}; 2{2}] nand [1{0},1{0}; 3{0}] nand [1{1},1{1}; 3{1}] nand [1{2},1{2}; 3{2}] nand [0{0},1{0}; 5{0}] nand [0{1},1{1}; 5{1}] nand [0{2},1{2}; 5{2}] nand [0{1}{0},0{1}{0}; 2{1}{0}] nand [0{1}{1},0{1}{1}; 2{1}{1}] nand [0{1}{2},0{1}{2}; 2{1}{2}] nand [1{1}{0},1{1}{0}; 3{1}{0}] nand [1{1}{1},1{1}{1}; 3{1}{1}] nand [1{1}{2},1{1}{2}; 3{1}{2}] nand [0{1}{0},1{1}{0}; 5{1}{0}] nand [0{1}{1},1{1}{1}; 5{1}{1}] nand [0{1}{2},1{1}{2}; 5{1}{2}]

nand [2,3; 4] nand [15(5{1}&5{2}),15(5{1}&5{2}); 16(61{1}&61{2})] nand [2{0},3{0}; 4{0}] nand [2{1},3{1}; 4{1}] nand [2{2},3{2}; 4{2}] nand [2{1}{0},3{1}{0}; 4{1}{0}] nand [2{1}{1},3{1}{1}; 4{1}{1}] nand [2{1}{2},3{1}{2}; 4{1}{2}] nand [15(5{1}&5{2}){0}(5{1}{0}),15(5{1}&5{2}){0}(5{1}{0}); 61{1}{0}] nand [15(5{1}&5{2}){1}(5{1}{1}&5{1}{2}),15(5{1}&5{2}){1}(5{1}{1}&5{1}{2}); 16(61{1}&61{2}){1}(61{1}{1}&61{1}{2})] nand [15(5{1}&5{2}){2}(5{2}),15(5{1}&5{2}){2}(5{2}); 61{2}] nand [5{1}{1},5{1}{1}; 61{1}{1}] nand [5{1}{2},5{1}{2}; 61{1}{2}]

nand [4,5; 6] nand [4{0},5{0}; 6{0}] nand [4{1},5{1}; 6{1}] nand [4{2},5{2}; 6{2}] nand [4{1}{0},5{1}{0}; 6{1}{0}] nand [4{1}{1},5{1}{1}; 6{1}{1}] nand [4{1}{2},5{1}{2}; 6{1}{2}]

nand [6,6; 7] nand [6{0},6{0}; 7{0}] nand [6{1},6{1}; 7{1}] nand [6{2},6{2}; 7{2}] nand [6{1}{0},6{1}{0}; 7{1}{0}] nand [6{1}{1},6{1}{1}; 7{1}{1}] nand [6{1}{2},6{1}{2}; 7{1}{2}] nand [6{1}{0},5{1}{1}; 69{1}{0}] nand [6{1}{1},5{1}{2}; 69{1}{1}]

nand [11(7{0}&7{1}),11(7{0}&7{1}); 33(6{0}&6{1})] nand [11(7{0}&7{1}),16(61{1}&61{2}); 41(73{0}&73{1})] nand [11(7{0}&7{1}){1}(7{1}{0}&7{1}{1}),11(7{0}&7{1}){1}(7{1}{0}&7{1}{1}); 33(6{0}&6{1}){1}(6{1}{0}&6{1}{1})] nand [11(7{0}&7{1}){0}(7{0}),16(61{1}&61{2}){0}(61{1}{0}); 73{0}] nand [11(7{0}&7{1}){1}(7{1}{0}&7{1}{1}),16(61{1}&61{2}){1}(61{1}{1}&61{1}{2}); 41(73{0}&73{1}){1}(73{1}{0}&73{1}{1})] nand [11(7{0}&7{1}){2}(7{1}{2}),16(61{1}&61{2}){2}(61{2}); 73{1}{2}] nand [7{1}{0},61{1}{1}; 73{1}{0}] nand [7{1}{1},61{1}{2}; 73{1}{1}]

nand [49(41(73{0}&73{1}){1}(73{1}{0}&73{1}{1})&41(73{0}&73{1}){2}(73{1}{2})),49(41(73{0}&73{1}){1}(73{1}{0}&73{1}{1})&41(73{0}&73{1}){2}(73{1}{2})); 50(91{1}(95{0}&95{1})&91{2})] nand [41(73{0}&73{1}){1}(73{1}{0}&73{1}{1}),41(73{0}&73{1}){1}(73{1}{0}&73{1}{1}); 91{1}(95{0}&95{1})] nand [41(73{0}&73{1}){2}(73{1}{2}),41(73{0}&73{1}){2}(73{1}{2}); 91{2}] nand [73{1}{0},73{1}{0}; 95{0}] nand [73{1}{1},73{1}{1}; 95{1}] nand [33(6{0}&6{1}){0}(6{0}),15(5{1}&5{2}){0}(5{1}{0}); 69{0}] nand [33(6{0}&6{1}),15(5{1}&5{2}); 45(69{0}&69{1})] nand [33(6{0}&6{1}){2}(6{1}{2}),15(5{1}&5{2}){2}(5{2}); 69{1}{2}] nand [33(6{0}&6{1}){1}(6{1}{0}&6{1}{1}),15(5{1}&5{2}){1}(5{1}{1}&5{1}{2}); 45(69{0}&69{1}){1}(69{1}{0}&69{1}{1})]

nand [45(69{0}&69{1}),41(73{0}&73{1}); 51(77{0}&77{1})] nand [45(69{0}&69{1}){0}(69{0}),41(73{0}&73{1}){0}(73{0}); 77{0}] nand [45(69{0}&69{1}){1}(69{1}{0}&69{1}{1}),41(73{0}&73{1}){1}(73{1}{0}&73{1}{1});

51(77{0}&77{1}){1}(77{1}{0}&77{1}{1})] nand
[45(69{0}&69{1}){2}(69{1}{2}),41(73{0}&73{1}){2}(73{1}{2}); 77{1}{2}] nand
[69{1}{0},73{1}{0}; 77{1}{0}] nand [69{1}{1},73{1}{1}; 77{1}{1}]

nand [51(77{0}&77{1}),51(77{0}&77{1}); 30(81{0}&81{1})] nand
[51(77{0}&77{1}){0}(77{0}),51(77{0}&77{1}){0}(77{0}); 81{0}] nand
[51(77{0}&77{1}){1}(77{1}{0}&77{1}{1}),51(77{0}&77{1}){1}(77{1}{0}&77{1}{1});
30(81{0}&81{1}){1}(81{1}{0}&81{1}{1})] nand
[51(77{0}&77{1}){2}(77{1}{2}),51(77{0}&77{1}){2}(77{1}{2}); 81{1}{2}] nand
[77{1}{0},77{1}{0}; 81{1}{0}] nand [77{1}{1},77{1}{1}; 81{1}{1}]

Y se fusionan los nodos.

addCopy[0,1,28; 18(17(28&30{2})&7{2}),55(30{0}&30{1}),50] =

nand [0,0; 2] nand [1,1; 3] nand [0,1; 5]

nand [2,3; 4] nand [15(5{1}&5{2}),15(5{1}&5{2}); 16]

nand [4,5; 6]

nand [6,6; 7]

nand [11(7{0}&7{1}),16; 41]

nand [49(41{1}&41{2}),49(41{1}&41{2}); 50] nand [33(6{0}&6{1}),15(5{1}&5{2}); 45]

nand [45,41; 51]

nand [51,51; 30]

De esta expresión, se extraen los nodos de entrada y salida de la nueva función recursiva, utilizando la tabla de equivalencia entre nodos expandidos y nodos originales de la definición de la función.

recursiveIn1:[16, 33(6{0}&6{1}), 15(5{1}&5{2}), 11(7{0}&7{1})]

recursiveOut1:[17(28&30{2})]

recursiveIn0:[1, 2, 3, 0]

recursiveOut0:[18(17&7{2})]

recursiveInInstance:[1, 3, 7, 0]

recursiveOutInstance:[29(2&24{2})]

Modificamos la definición original de la suma con la instancia a esta nueva función recursiva.

add[0,1; 10] =

nand [0,0; 2] nand [1,1; 6]

addRecur [1,2,6,0; 10]

Y calculamos la nueva función recursiva.

addRecur[0,1,2,3; 21(22&16{2})] =

nand [3,0; 4] nand [1,2; 10]

nand [7(4{1}&4{2}),7(4{1}&4{2}); 9] nand [10,4; 11]

nand [11,11; 16]

addRecur [9,13(11{0}&11{1}),7(4{1}&4{2}),18(16{0}&16{1}); 22]

Con lo que ya tenemos el algoritmo de la suma modular optimizado.

7.7 Traza de ejecución del software de optimización desarrollado

El desarrollo del software de optimización ha sido incremental, tal y como refleja el crecimiento de la base de datos durante su ejecución.

Se comenzó por expresar, almacenar y optimizar funciones secuenciales extremadamente sencillas.

A partir de ellas, se fueron definiendo instrucciones más complejas, incluyendo funciones recursivas.

Finalmente se implementó el algoritmo de la suma y se optimizó.

```
[00001111, 00110011]nand[11111100]
Definition as nands:
nandCopy[0,1; 2] =
       root in:
Definition cost in parallel nands: 0

New definition:
not[0; 1] =
       nand [0,0;1]
       root in:
Definition as nands:
notCopy[0; 1] =
       nand [0,0;1]
       root in:
Definition cost in parallel nands: 1
[00001111]not[11110000]
Optimized definition:
not[0; 1] =
       nand [0,0;1]
```

```
       root in:
Definition as nands:
notCopy[0; 1] =
       nand [0,0;1]
       root in:
Definition cost in parallel nands: 1
[00001111]not[11110000]
DATABASE:
not[0; 1] =
       nand [0,0;1]
       root in:
nand[0,1; 2] =
       root in: not,

New definition:
and[0,1; 2] =
       nand [0,1;3]
       not [3;2]
       root in:
Definition as nands:
andCopy[0,1; 2] =
       nand [0,1;3]
       nand [3,3;2]
       root in:
Definition cost in parallel nands: 2
[00001111, 00110011]and[00000011]
Optimized definition:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
Definition as nands:
andCopy[0,1; 2] =
       nand [0,1;3]
       nand [3,3;2]
       root in:
Definition cost in parallel nands: 2
[00001111, 00110011]and[00000011]
DATABASE:
not[0; 1] =
       nand [0,0;1]
       root in:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
nand[0,1; 2] =
       root in: not,and,

New definition:
or[0,1; 2] =
       not [0;3] not [1;4]
       nand [3,4;2]
       root in:
Definition as nands:
orCopy[0,1; 2] =
       nand [0,0;3] nand [1,1;4]
       nand [3,4;2]
       root in:
Definition cost in parallel nands: 2
[00001111, 00110011]or[00111111]
```

```
Optimized definition:
or[0,1; 4] =
       nand [0,0;2] nand [1,1;3]
       nand [2,3;4]
       root in:
Definition as nands:
orCopy[0,1; 2] =
       nand [0,0;3] nand [1,1;4]
       nand [3,4;2]
       root in:
Definition cost in parallel nands: 2
[00001111, 00110011]or[00111111]
DATABASE:
not[0; 1] =
       nand [0,0;1]
       root in:
or[0,1; 4] =
       nand [0,0;2] nand [1,1;3]
       nand [2,3;4]
       root in:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
nand[0,1; 2] =
       root in: not,and,or,

New definition:
xor[0,1; 2] =
       nand [0,1;3]
       nand [3,0;4] nand [3,1;5]
       nand [4,5;2]
       root in:
Definition as nands:
xorCopy[0,1; 2] =
       nand [0,1;4]
       nand [4,0;3] nand [4,1;5]
       nand [3,5;2]
       root in:
Definition cost in parallel nands: 3
[00001111, 00110011]xor[00111100]
Optimized definition:
xor[0,1; 5] =
       nand [0,1;2]
       nand [2,0;3] nand [2,1;4]
       nand [3,4;5]
       root in:
Definition as nands:
xorCopy[0,1; 2] =
       nand [0,1;4]
       nand [4,0;3] nand [4,1;5]
       nand [3,5;2]
       root in:
Definition cost in parallel nands: 3
[00001111, 00110011]xor[00111100]
DATABASE:
not[0; 1] =
       nand [0,0;1]
       root in:
or[0,1; 4] =
       nand [0,0;2] nand [1,1;3]
```

```
       nand [2,3;4]
       root in:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
nand[0,1; 2] =
       root in: not,and,or,xor,
xor[0,1; 5] =
       nand [0,1;2]
       nand [2,0;3] nand [2,1;4]
       nand [3,4;5]
       root in:

New definition:
if[0,1,2; 3] =
       not [0;4] and [0,2;6]
       and [4,1;5]
       or [5,6;3]
       root in:
Definition as nands:
ifCopy[0,1,2; 3] =
       nand [0,0;5] nand [0,2;8]
       nand [5,1;7] nand [8,8;6]
       nand [7,7;4] nand [6,6;10]
       nand [4,4;9]
       nand [9,10;3]
       root in:
Definition cost in parallel nands: 5
[00001111, 00110011, 01010101]if[00110101]
Optimized definition:
if[0,1,2; 6] =
       nand [0,0;3] nand [0,2;5]
       nand [3,1;4]
       nand [4,5;6]
       root in:
Definition as nands:
ifCopy[0,1,2; 3] =
       nand [0,0;5] nand [0,2;6]
       nand [5,1;4]
       nand [4,6;3]
       root in:
Definition cost in parallel nands: 3
[00001111, 00110011, 01010101]if[00110101]
DATABASE:
not[0; 1] =
       nand [0,0;1]
       root in:
or[0,1; 4] =
       nand [0,0;2] nand [1,1;3]
       nand [2,3;4]
       root in:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
nand[0,1; 2] =
       root in: not,and,or,xor,if,
xor[0,1; 5] =
       nand [0,1;2]
       nand [2,0;3] nand [2,1;4]
```

```
       nand [3,4;5]
       root in:
if[0,1,2; 6] =
       nand [0,0;3] nand [0,2;5]
       nand [3,1;4]
       nand [4,5;6]
       root in:

New definition:
rif[0,1,2; 3(8&9)] =
       rif [0,1{1..n-1},2{1..n-1};8] if [0,1{n},2{n};9]
       root in:
Definition as nands:
rifCopy[0,1,2; 3(9&4)] =
       rifCopy [0,1{1..n-1},2{1..n-1};9] nand [0,0;10] nand
[0,2{n};12]
       nand [10,1{n};11]
       nand [11,12;4]
       root in:
Definition cost in parallel nands: 3
[1, 00110011, 01010101]rif[01010101]
Optimized definition:
rif[0,1,2; 3(8&9)] =
       rif [0,1{1..n-1},2{1..n-1};8] nand [0,0;10] nand [0,2{n};12]
       nand [10,1{n};11]
       nand [11,12;9]
       root in:
Definition as nands:
rifCopy[0,1,2; 3(12&4)] =
       rifCopy [0,1{1..n-1},2{1..n-1};12] nand [0,0;6] nand [0,2{n};9]
       nand [6,1{n};5]
       nand [5,9;4]
       root in:
Definition cost in parallel nands: 3
[1, 00110011, 01010101]rif[01010101]
DATABASE:
not[0; 1] =
       nand [0,0;1]
       root in:
or[0,1; 4] =
       nand [0,0;2] nand [1,1;3]
       nand [2,3;4]
       root in:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
nand[0,1; 2] =
       root in: not,and,or,xor,if,
xor[0,1; 5] =
       nand [0,1;2]
       nand [2,0;3] nand [2,1;4]
       nand [3,4;5]
       root in:
if[0,1,2; 6] =
       nand [0,0;3] nand [0,2;5]
       nand [3,1;4]
       nand [4,5;6]
       root in:
rif[0,1,2; 3(8&9)] =
       rif [0,1{1..n-1},2{1..n-1};8] nand [0,0;10] nand [0,2{n};12]
```

```
       nand [10,1{n};11]
       nand [11,12;9]
       root in:

New definition:
sum[0,1,2; 3,4] =
       xor [0,1;5] and [0,1;6] xor [0,1;7]
       and [5,2;8] xor [7,2;3]
       or [6,8;4]
       root in:
Definition as nands:
sumCopy[0,1,2; 3,5] =
       nand [0,1;9] nand [0,1;15] nand [0,1;16]
       nand [9,0;10] nand [9,1;11] nand [15,15;6] nand [16,0;17] nand
[16,1;18]
       nand [10,11;4] nand [17,18;8] nand [6,6;20]
       nand [4,2;12] nand [8,2;19]
       nand [12,4;13] nand [12,2;14] nand [19,19;7]
       nand [13,14;3] nand [7,7;21]
       nand [20,21;5]
       root in:
Definition cost in parallel nands: 7
[00001111, 00110011, 01010101]sum[01101001, 00010111]
Optimized definition:
sum[0,1,2; 10,11] =
       nand [0,1;3]
       nand [3,0;4] nand [3,1;5]
       nand [4,5;6]
       nand [6,2;7]
       nand [7,6;8] nand [7,2;9] nand [3,7;11]
       nand [8,9;10]
       root in:
Definition as nands:
sumCopy[0,1,2; 3,11] =
       nand [0,1;8]
       nand [8,0;7] nand [8,1;9]
       nand [7,9;6]
       nand [6,2;5]
       nand [5,6;4] nand [5,2;10] nand [8,5;11]
       nand [4,10;3]
       root in:
Definition cost in parallel nands: 6
[00001111, 00110011, 01010101]sum[01101001, 00010111]
DATABASE:
not[0; 1] =
       nand [0,0;1]
       root in:
or[0,1; 4] =
       nand [0,0;2] nand [1,1;3]
       nand [2,3;4]
       root in:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
nand[0,1; 2] =
       root in: not,and,or,xor,if,sum,
xor[0,1; 5] =
       nand [0,1;2]
       nand [2,0;3] nand [2,1;4]
       nand [3,4;5]
```

```
       root in:
sum[0,1,2; 10,11] =
       nand [0,1;3]
       nand [3,0;4] nand [3,1;5]
       nand [4,5;6]
       nand [6,2;7]
       nand [7,6;8] nand [7,2;9] nand [3,7;11]
       nand [8,9;10]
       root in:
if[0,1,2; 6] =
       nand [0,0;3] nand [0,2;5]
       nand [3,1;4]
       nand [4,5;6]
       root in:
rif[0,1,2; 3(8&9)] =
       rif [0,1{1..n-1},2{1..n-1};8] nand [0,0;10] nand [0,2{n};12]
       nand [10,1{n};11]
       nand [11,12;9]
       root in:

New definition:
sumR[0,1,2; 3(5&6),4(7&8)] =
       sum [0{n},1{n},2;6,7]
       sumR [0{1..n-1},1{1..n-1},7;5,8]
       root in:
Definition as nands:
sumRCopy[0,1,2; 3(10&4),12] =
       nand [0{n},1{n};13]
       nand [13,0{n};14] nand [13,1{n};15]
       nand [14,15;16]
       nand [16,2;17]
       nand [17,16;18] nand [17,2;19] nand [13,17;9(12&]
       nand [18,19;4] sumRCopy [0{1..n-1},1{1..n-1},9(12&;10,1112)]
       root in:
Definition cost in parallel nands: 0+6x^1
[00001111, 00110011, 1]sumR[01000011, 11111100]
Optimized definition:
sumR[0,1,2; 3(5&6),4(7&8)] =
       nand [0{n},1{n};13]
       nand [13,0{n};14] nand [13,1{n};15]
       nand [14,15;16]
       nand [16,2;17]
       nand [17,16;18] nand [17,2;19] nand [13,17;7]
       nand [18,19;6] sumR [0{1..n-1},1{1..n-1},7;5,8]
       root in:
Definition as nands:
sumRCopy[0,1,2; 3(16&4),19] =
       nand [0{n},1{n};9]
       nand [9,0{n};8] nand [9,1{n};14]
       nand [8,14;7]
       nand [7,2;6]
       nand [9,6;17(19&] nand [6,7;5] nand [6,2;15]
       sumRCopy [0{1..n-1},1{1..n-1},17(19&;16,1819)] nand [5,15;4]
       root in:
Definition cost in parallel nands: 0+5x^1
[00001111, 00110011, 1]sumR[01000011, 11111100]
DATABASE:
sumR[0,1,2; 3(5&6),4(7&8)] =
       nand [0{n},1{n};13]
       nand [13,0{n};14] nand [13,1{n};15]
       nand [14,15;16]
```

```
       nand [16,2;17]
       nand [17,16;18] nand [17,2;19] nand [13,17;7]
       nand [18,19;6] sumR [0{1..n-1},1{1..n-1},7;5,8]
       root in:
not[0; 1] =
       nand [0,0;1]
       root in:
or[0,1; 4] =
       nand [0,0;2] nand [1,1;3]
       nand [2,3;4]
       root in:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
nand[0,1; 2] =
       root in: not,and,or,xor,if,sum,
xor[0,1; 5] =
       nand [0,1;2]
       nand [2,0;3] nand [2,1;4]
       nand [3,4;5]
       root in:
sum[0,1,2; 10,11] =
       nand [0,1;3]
       nand [3,0;4] nand [3,1;5]
       nand [4,5;6]
       nand [6,2;7]
       nand [7,6;8] nand [7,2;9] nand [3,7;11]
       nand [8,9;10]
       root in:
if[0,1,2; 6] =
       nand [0,0;3] nand [0,2;5]
       nand [3,1;4]
       nand [4,5;6]
       root in:
rif[0,1,2; 3(8&9)] =
       rif [0,1{1..n-1},2{1..n-1};8] nand [0,0;10] nand [0,2{n};12]
       nand [10,1{n};11]
       nand [11,12;9]
       root in:

New definition:
add[0,1; 2(7(9{n}&10)&11)] =
       and [0{n},1{n};12] xor [0{n},1{n};11]
       sumR [0{1..n-1},1{1..n-1},12;10,9]
       root in:
Definition as nands:
addCopy[0,1; 2(8(11{n}&9)&3)] =
       nand [0{n},1{n};13] nand [0{n},1{n};14]
       nand [13,13;10] nand [14,0{n};15] nand [14,1{n};16]
       sumR [0{1..n-1},1{1..n-1},10;9,11] nand [15,16;3]
       root in:
Definition cost in parallel nands: 0+5x^1
[00001111, 00110011]add[001000010]
[1, 01]add[010]
Optimized definition:
add[0,1; 2(7(9{n}&10)&11)] =
       nand [0{n},1{n};13]
       nand [13,13;12] nand [13,0{n};15] nand [13,1{n};16]
       sumR [0{1..n-1},1{1..n-1},12;10,9] nand [15,16;11]
       root in:
```

```
Definition as nands:
addCopy[0,1; 2(11(15{n}&12)&3)] =
       nand [0{n},1{n};14]
       nand [14,14;13] nand [14,0{n};4] nand [14,1{n};10]
       sumR [0{1..n-1},1{1..n-1},13;12,15] nand [4,10;3]
       root in:
Definition cost in parallel nands: 0+5x^1
[00001111, 00110011]add[001000010]
[1, 01]add[010]
DATABASE:
add[0,1; 2(7(9{n}&10)&11)] =
       nand [0{n},1{n};13]
       nand [13,13;12] nand [13,0{n};15] nand [13,1{n};16]
       sumR [0{1..n-1},1{1..n-1},12;10,9] nand [15,16;11]
       root in:
sumR[0,1,2; 3(5&6),4(7&8)] =
       nand [0{n},1{n};13]
       nand [13,0{n};14] nand [13,1{n};15]
       nand [14,15;16]
       nand [16,2;17]
       nand [17,16;18] nand [17,2;19] nand [13,17;7]
       nand [18,19;6] sumR [0{1..n-1},1{1..n-1},7;5,8]
       root in:
not[0; 1] =
       nand [0,0;1]
       root in:
or[0,1; 4] =
       nand [0,0;2] nand [1,1;3]
       nand [2,3;4]
       root in:
and[0,1; 3] =
       nand [0,1;2]
       nand [2,2;3]
       root in:
nand[0,1; 2] =
       root in: not,and,or,xor,if,sum,
xor[0,1; 5] =
       nand [0,1;2]
       nand [2,0;3] nand [2,1;4]
       nand [3,4;5]
       root in:
sum[0,1,2; 10,11] =
       nand [0,1;3]
       nand [3,0;4] nand [3,1;5]
       nand [4,5;6]
       nand [6,2;7]
       nand [7,6;8] nand [7,2;9] nand [3,7;11]
       nand [8,9;10]
       root in:
if[0,1,2; 6] =
       nand [0,0;3] nand [0,2;5]
       nand [3,1;4]
       nand [4,5;6]
       root in:
rif[0,1,2; 3(8&9)] =
       rif [0,1{1..n-1},2{1..n-1};8] nand [0,0;10] nand [0,2{n};12]
       nand [10,1{n};11]
       nand [11,12;9]
       root in:
```

8 Figuras

9 Glosario

Literal: en lógica matemática, un literal es una fórmula atómica o su negación.

Síntesis de un circuito: proceso por el cual se implementa mediante puertas lógicas el diseño de un circuito. Se utiliza como sinónimo de minimización del circuito. [5]

Red booleana: modelo estándar independiente de la tecnología para representar circuitos mediante una red lógica. Los nodos de la red pueden ser entradas primarias, salidas primarias o funciones booleanas. Las funciones representan una expresión arbitraria, con un número ilimitado de entradas locales y una única salida. [12]

Bosque: estructura de datos arborescente compuesta de varios árboles de nodos.

Profundidad: cantidad de funciones booleanas de una Red Booleana que han de ejecutarse necesariamente en un orden secuencial. Es equivalente al coste secuencial de ejecución si asumimos la ejecución con la máxima paralelización posible.

Algoritmo de Quine-McCluskey: método clásico de minimización de funciones booleanas. El resultado es una suma de productos (OR de ANDs).

Minimización en dos niveles: minimización a una expresión como suma de productos (OR de ANDs) o producto de sumas (AND de ORs). El objetivo es reducir el número de literales y productos (o sumas) utilizados.

Minimización multinivel: minimización a una expresión con funciones lógicas arbitrarias. El objetivo es reducir el número de literales utilizados.

Algoritmo Espresso: algoritmo heurístico de minimización de circuitos en dos niveles que es el estándar de facto para la industria.

Algoritmo Espresso-exacto o "**mincov**" (mínimum-covering): algoritmo exacto de minimización de circuitos. Pese al nombre, no tiene relación con el algoritmo Espresso y es una implementación moderna del método de Quine-McCluskey. [15]

MisII: herramienta desarrollada en la Universidad de California, Berkeley para realizar síntesis multinivel de circuitos. Utiliza Espresso como una de sus subrutinas. [5] [16]

Logic Friday: software para usar los métodos Espresso, Espresso-exacto y MisII de minimización de circuitos.

Algoritmo NP: algoritmos de complejidad de tiempo polinómico no determinista.

Algoritmo NP-completo: algoritmos más difíciles de NP.

Algoritmo NP-complejo: algoritmos que son como mínimo tan difíciles como un problema de NP.

Algoritmo $\sum_2^P$: algoritmos más complejos que NP. Serían NP si tuviéramos acceso a una máquina oráculo que resolviera un problema NP cuantas veces queramos.

Algoritmo $\sum_2^P$ −completo: dentro de la jerarquía de clases de complejidad, los algoritmos más difíciles de $\sum_2^P$.

Búsqueda en anchura: algoritmo para explorar un grafo, explorando primero los vecinos de los nodos.

Búsqueda en profundidad: algoritmo para explorar un grafo siguiendo un camino concreto, expandiendo cada nodo que se va localizando.

Padre: en un árbol, nodo con hijos. En este documento se utiliza para mantener un orden: un padre siempre está más cerca de la raíz (o de las entradas) que los hijos.

Hijo: en un árbol, nodo descendiente de otro nodo. En este documento se utiliza para mantener un orden: los hijos siempre están más cerca de las hojas (o salidas) que su padre.

Supernodo: nodo formado por múltiples nodos en una estructura que no tiene porqué ser un árbol. Se utiliza en este documento para diferenciar cuando no se sigue una dirección concreta como con un nodo Padre.

Subnodo: nodo que es parte de un supernodo en una estructura que no tiene porqué ser un árbol. Se utiliza en este documento para diferenciar cuando no se sigue una dirección concreta como con un nodo hijo.

Tabla dispersa: estructura de datos que asocia claves con valores y cuyo tiempo de acceso es constante en media.

Funciones μ-recursivas: En matemáticas y ciencias de la computación las funciones μ-recursivas son una clase de funciones parciales de los números naturales a los números naturales que son "computables" en un sentido intuitivo.