Proyecto de Grado 2014

Generación Aleatoria de Funciones Inmunes a la Correlación de menor peso

Sebastián Fonseca
Ma. Cecilia García
Tutor: Alfredo Viola

CONTENIDO

RESUMEN

Trabajos realizados por Le Bars y Viola (Le Bars, y otros, 2010), así como también por Carrasco (Carrasco, et al., 2011) han propuesto algoritmos tanto de generación recursiva de funciones inmunes a la correlación de primer orden, como algoritmos de enumeración de dichas funciones. En este proyecto se diseñará e implementará de manera eficiente la generalización de dichos algoritmos. A su vez, se presentan algoritmos para la generación de clases de equivalencia inmunes a la correlación de orden mayor que uno y generación aleatoria de las funciones correspondientes. Estos algoritmos podrán utilizarse para funciones de alta cantidad de variables y orden de correlación. Además, se brinda información adicional relativa a la cantidad de funciones por clases y el detalle de la construcción de las clases de equivalencia, así como también la posibilidad de estudiar patrones en la extensión de las mismas.

CAPÍTULO 1 – INTRODUCCIÓN

En la actualidad las funciones Booleanas (funciones de n variables tales que $f_n: \{0,1\}^n \rightarrow \{0,1\}$), son empleadas en sistemas criptográficos. Las mismas deben cumplir algunas propiedades importantes para ser resistentes a ciertos ataques, tales como alto grado algebraico o ser inmunes a la correlación.

Por ejemplo, en la prevención de ataques que utilizan algoritmos de Berlekam-Massey (Massey, 1969) se emplean funciones con un alto grado algebraico. Otro ejemplo es el ataque de correlación de Siegenthaler (Siegenthaler, 1984) para el cual es indicado utilizar funciones inmunes a la correlación.

DEFINICIÓN 1.1 – INMUNE A LA CORRELACIÓN
Una función Booleana de n variables es inmune a la correlación de orden k si fijando cualquier conjunto de k entradas todas las 2^k sub-funciones tienen el mismo peso de Hamming, es decir, misma cantidad de 1's en su tabla de verdad.

Un relevamiento muy completo del uso de funciones Booleanas en criptografía y códigos de corrección de errores es presentado por Carlet en varios artículos (Carlet 2010 A, Carlet 2010 B). Por otra parte, Palmer, Read y Robinson (E.M. Palmer, 1992) plantearon dos métodos para el conteo de funciones Booleanas 1-resiliente (balanceadas e inmunes a la correlación) obteniendo el número para funciones de hasta 6 variables. Posteriormente Strazdins (Strazdins, 1997) investigó la clasificación de las funciones Booleanas mediante clases de equivalencia, mostrando que las mismas pueden ser organizadas secuencialmente.

Dado que los métodos anteriormente citados no son factibles de emplearse en funciones Booleanas de gran número de variables porque se genera un problema de explosión combinatoria, se debe utilizar una representación de las funciones que las agrupe en clases de equivalencias.

El primer nivel de agrupación de funciones Booleanas es mediante la definición de clases de equivalencia de funciones; estas se pueden definir de varias formas, Carlet explica que las clases de equivalencia de funciones inmunes a la correlación de orden k pueden ser definidas utilizando la propiedad de la transformada de Walsh (Carlet, 2010 A).

Al re-definir el problema del conteo y generación aleatoria de funciones Booleanas en base a clases de equivalencia se deben definir algoritmos que trabajen con clases y que luego sea posible obtener las funciones a partir de los datos generados.

En (Le Bars, y otros, 2010) se prueba que es necesario contar con cada función de n-2 variables para poder construir todas las funciones 1 resilientes de n variables. En (Carrasco, et al., 2011) por su parte se continúa el trabajo anteriormente citado brindando algoritmos prácticos para la generación aleatoria uniforme de funciones inmunes a la correlación de primer orden.

Este proyecto extiende los trabajos de (Le Bars, y otros, 2010) y (Carrasco, et al., 2011) presentando algoritmos eficientes para el cálculo de clases de equivalencia de funciones inmunes a la correlación de peso mínimo y generación aleatoria uniforme de las funciones correspondientes probados y evaluados para funciones de hasta 13 variables y orden de correlación 13.

1.1 Definición del problema

El desarrollo del proyecto constó de varias etapas, en principio se relevó el estado del arte de los trabajos realizados sobre funciones Booleanas inmunes a la correlación y además se estudió lo que correspondía con el marco teórico relacionado al problema. Luego de haber profundizado en ambos temas se decidió diseñar e implementar algoritmos que avanzaran en el problema. Para ello y siguiendo las ideas presentadas en (Carrasco, et al., 2011) y (Le Bars, y otros, 2010) se optó por agrupar las funciones en clases de equivalencia. Por lo tanto, el objetivo del algoritmo fue hallar las clases de equivalencia que representaran funciones Booleanas inmunes a la correlación de peso de Hamming mínimo.

El proyecto consta de dos partes; por un lado, el diseño e implementación de algoritmos para obtener todas las funciones Booleanas con menor peso de Hamming inmunes a la correlación de cierto orden k y por otro la generación aleatoria uniforme de cada una de estas funciones.

Parte 1 - Funciones Booleanas inmunes a la correlación de orden k de menor peso de Hamming

Esta primera parte consta del diseño e implementación de un algoritmo eficiente que resuelva la siguiente problemática: dados dos enteros *n* y *k*, generar todas las funciones de *n* variables inmunes a la correlación de orden *k* con menor peso de Hamming.

Para resolver este problema es necesario ir calculando todas las funciones de *n* variables con cierto peso de Hamming mayor a cero y verificar si alguna de ellas es inmune a la correlación de orden *k*, partiendo por funciones Booleanas con pesos de Hamming pequeños. Este proceso es incremental tanto en *n* como en el peso para el cual se está buscando una posible solución.

Se podrá ver más adelante en la sección de 2.1 Estado del Arte, que la cantidad de funciones que estamos trabajando crece de manera doblemente exponencial por lo tanto es imprescindible poder hallar una representación eficiente del problema para obtener buenos resultados.

Dadas estas enormes cantidades de funciones se optó por utilizar la agrupación de funciones Booleanas mediante clases de equivalencia, según lo presentado en (Carrasco, et al., 2011) y (Le Bars, y otros, 2010).

Dos funciones pertenecen a la misma clase si tienen las mismas propiedades combinatorias en relación a este problema. En el Marco teórico se va a formalizar este concepto. Esta manera de resolver el problema es más eficiente que operar con todas las funciones Booleanas, ya que una clase de equivalencia representa a un conjunto de funciones Booleanas y una función Booleana pertenece sólo a una clase de equivalencia.

Con esta modificación, el algoritmo debe encontrar la clase de equivalencia que representa al conjunto de funciones Booleanas de *n* variables inmunes a la correlación de orden *k* de menor peso de Hamming.

Al obtenerse la clase de equivalencia se podrá calcular la cantidad de funciones pertenecientes a ella, así como también obtener el listado completo de las funciones pertenecientes a la clase, según las generalizaciones de los algoritmos presentados (Carrasco, et al., 2011) y (Le Bars, y otros, 2010)

PARTE 2 - GENERACIÓN ALEATORIA DE FUNCIONES BOOLEANAS INMUNES A LA CORRELACIÓN DE MENOR PESO DE HAMMING

El método de las clases presentado en (Le Bars, y otros, 2010) permite caracterizar todas las funciones inmunes a la correlación y contarlas. Como se ve en (Carrasco, et al., 2011) esta caracterización permite además generar aleatoriamente y de manera uniforme todas las funciones Booleanas pertenecientes a una clase dada. Este proyecto generaliza las ideas presentadas en (Carrasco, et al., 2011) y (Le Bars, y otros, 2010) para $k \geq 1$.

El informe se divide de la siguiente manera: en la sección 2 presentamos la motivación del problema y las definiciones más importantes. En la sección 3 describimos el proyecto y sus principales características, mostramos como se validaron los resultados obtenidos e ilustramos con algunos de ellos.

Posteriormente, en la sección 4 describimos la implementación realizada para los algoritmos diseñados, así como también la tecnología utilizada, las consideraciones tomadas para la elección del lenguaje de programación y base de datos.

En la sección 5 analizamos los resultados obtenidos y planteamos una búsqueda de simetrías en las clases de equivalencia. Las conclusiones obtenidas y la presentación de algunos trabajos a futuro para seguir extendiendo la investigación son desarrollados en la sección 6.

Mediante los anexos se complementa todo lo descripto en las secciones anteriores: en el Anexo I se presenta el manual de la aplicación junto con ejemplos de ejecución de la misma. En el Anexo II mostramos cálculos manuales de clases de equivalencias y sus respectivas funciones Booleanas. En el Anexo III exponemos algunas de las funciones Booleanas obtenidas con la aplicación. En el Anexo IV se exhiben algunos casos interesantes de extensiones de clases de equivalencia en un grado más con la intensión de encontrar simetrías en las mismas y por último en el Anexo V se presentan los datos correspondientes a la gestión del proyecto.

1.2 MOTIVACIÓN

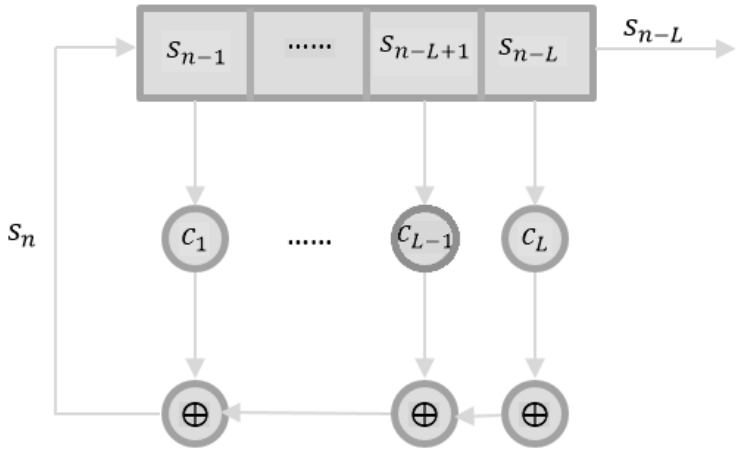
USO DE FUNCIONES BOOLEANAS EN CRIPTOGRAFÍA

Las funciones Booleanas son muy utilizadas tanto en criptografía como en códigos de corrección de errores (Carlet, 2010 A).

En cuanto al enfoque criptográfico, el rol que cumplen las funciones Booleanas es también importante. Algunas transformaciones criptográficas como los generadores pseudo aleatorios en cifrados por flujo y "S-boxes" en cifrados por bloque son diseñadas usando composiciones apropiadas de funciones Booleanas no lineales.

El cifrado por flujo se realiza incrementalmente convirtiendo el texto en claro en un texto cifrado bit a bit. Esto es posible mediante un generador de flujo de clave. Este flujo es una secuencia de bits de cierto tamaño y puede utilizarse para encriptar el contenido de datos combinando el flujo de clave con el flujo de datos mediante la función XOR. Es posible crear un generador de flujo de clave realizando iteraciones de una función matemática sobre un rango de valores de entrada.

Un mecanismo muy utilizado para generar una secuencia pseudo aleatoria a partir de una clave secreta es mediante LFSR (Linear Feedback Shift Registers). El comportamiento de un LFSR se ilustra en la siguiente imagen.

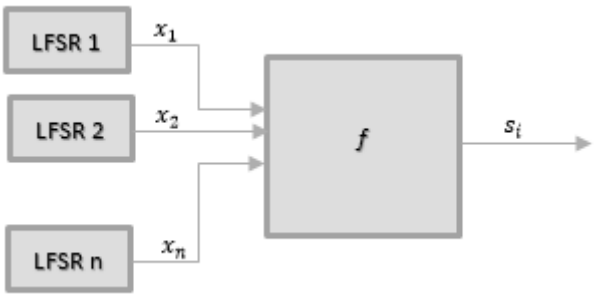


En cada ciclo de reloj, los bits $s_{n-1}, \dots, s_{n-L}$ contenidos en los flip-flops del LFSR se mueven a la derecha. El flip-flop de más a la izquierda es alimentado por la combinación lineal $\bigoplus_{i=1}^{L} c_i s_{n-i}$. Por lo tanto, cada salida recurrente del LFSR satisface la relación:

$$s_n = \bigoplus_{i=1}^{L} c_i s_{n-i}$$

Las secuencias generadas son siempre periódicas con período de a lo sumo 2^L-1; además la clave secreta (pequeña) es la inicialización $s_0, \dots, s_{L-1}$ del LFSR y genera los valores de los coeficientes de retorno, c_i.

Los LFSR pueden suponer una debilidad criptográfica, ya que por el algoritmo de Berlekamp-Massey (Massey, 1969) se puede llegar a recuperar la clave (que es el valor inicial o semilla) de manera eficiente. A continuación, vamos a ver que se pueden utilizar funciones Booleanas inmunes a la correlación para contrarrestar el uso de este algoritmo, ya que cada salida de los LFSR puede ser interpretada como las entradas a una función Booleana.



Para fijar ideas, vamos a analizar este caso con 3 entradas (3 LFSR's) y 3 salidas distintas (*f*).

x_3	x_2	x_1	f_1	f_2	f_3
0	0	0	0	1	0
0	0	1	0	1	1
0	1	0	1	0	1
0	1	1	1	0	0
1	0	0	0	0	1
1	0	1	0	0	0
1	1	0	1	1	0
1	1	1	1	1	1

Podemos ver para la función Booleana f_1 que, si sabemos el valor de la entrada x_2, sabemos cuál es la salida de la función Booleana, esto es $f_1(x_3, 0, x_1)$ = 0 y $f_1(x_3, 1, x_1)$ = 1 $\forall\ x_1, x_3 \in \{0,1\}$. Veremos más adelante que la función f_1 no es inmune a la correlación.

La función Booleana f_2 es inmune a la correlación de orden 1 pero no de orden 2, esto hace que sabiendo una entrada cualquiera de la función Booleana $x_i = j$, la taba de verdad de la función f_2 condicionada a $x_i = j$ tiene el mismo peso de Hamming para todo $1 \leq i \leq 3, j \in \{0,1\}$.

Al no ser de orden 2 y tener 3 variables, podemos hallar combinaciones de a 2 variables que nos permitan predecir la salida. Por ejemplo, para $f_2(0,0,x_1)$ la salida de f_2 siempre es 1 $\forall\ x_1 \in \{0,1\}$.

La última función (f_3) es inmune a la correlación de orden 2. Esto implica que también sea inmune a la correlación de orden 1. Por ejemplo $f_3(0,0,x_1)$ la salida de f_3 puede ser 0 o 1, pasa lo mismo para $f_3(0,1,x_1)$, es decir que la salida también tiene peso de Hamming 1.

Por otro lado, el cifrado por bloque es un algoritmo determinístico que opera sobre un conjunto de bits de largo fijo (bloque) con una transformación invariante que está especificada por la clave simétrica. Esto lo podemos ver aplicado en las S-Boxes que se utilizan, tanto para AES (Advanced Encryption Standar) como para DES (Data Encryption Standar).

Una S-Box (Substitution Box) es un componente básico en los algoritmos de clave simétrica que realiza una substitución. En los cifrados por bloque son utilizados típicamente para oscurecer la relación que hay entre la clave y el texto cifrado, es lo que Shannon (Shannon, 1949) describe como confusión.

En una S-Box se toma como entrada un número *m* de bits y los transforma en *n* de bits de salida, dónde *n* y *m* no tienen por qué ser iguales. Ejemplo de esto es función vectorial definida como $F_{2^n} \rightarrow F_{2^m}$ (Carlet, 2010 B); las funciones Booleanas son funciones vectoriales con m=1.

	x0000x	x0001x	x0010x	x0011x	x0100x	0x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0yyyy1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
1yyyy0	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
1yyyy1	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S-Box 1 de DES (NIST, 1999)

Por ejemplo, en la S-Box 1 de DES mostrada anteriormente, si tenemos como entrada 000000 la salida va a ser 00010100 (14 en hexadecimal).

A continuación, vamos a describir el algoritmo AES (Stinson, 2006) para ver cómo se puede llegar a utilizar funciones Booleanas inmunes a la correlación en la generación de las S-Boxes utilizadas.

AES manipula bloques de 128 bits y permite claves de 128 bits, 192 bits y 256 bits. Es un cifrado iterativo en el cual se maneja cada iteración como el número de ronda (N_r) y depende del largo de la clave. Si la clave es de 128 bits entonces N_r=10, si la clave es de 192 bits entonces N_r= 12 y si la clave es de 256 bits entonces N_r= 14.

El algoritmo AES básicamente cumple con los siguientes pasos:

1. Dado un texto plano x, inicializamos *State* con x y realizamos la operación ADD-ROUNDKEY que realiza X-OR's entre la clave de la Ronda y *State*.
2. Para cada uno de las N_r-1 rondas, realizar la operación de substitución llamada SUBBYTES a *State* usando una S-Box, además se debe realizar una permutación SHIFTROWS a *State*, luego se debe realizar la operación MIXCOLUMNS a *State* y por último realizamos la operación ADD-ROUNDKEY.
3. Realizar una vez más SUBBYTES, SHIFTROWS, MIXCOLUMNS y ADD-ROUNDKEY.
4. El texto cifrado y es *State*.

x\y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	F3	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8ª
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

S-Box AES, tanto entradas como salidas en notación hexadecimal (Stinson, 2006)

En DES las S-Boxes están predefinidas, pero en AES pueden ser definidas de manera algebraica. La formulación de una S-Box para AES involucra operaciones en un cuerpo finito, con lo cual, por ejemplo, podemos definir la S-Box como:

$$\mathrm{F}_{2^8} = \mathbb{Z}_2[x]/(x^8 + x^4 + x^3 + x + 1)$$

En ambos contextos, y por una razón de eficiencia, la cantidad de variables (*n*) no es muy grande. En los cifrados por flujo por lo general se utilizan 10 variables y para las "S-boxes" utilizadas en la mayoría de los cifrados por bloques, que son concatenaciones de sub-"S-boxes", como mucho se utilizan 8 variables.

Viendo estos valores destacamos que las cantidades de funciones Booleanas que están en juego son extremadamente grandes: ¡si la cantidad de variables es 8 entonces la cantidad de funciones Booleanas es aproximadamente la cantidad estimada de átomos del Universo!

En la siguiente tabla se muestra la cantidad de funciones Booleanas $|BF_n|$ para n de 4 a 8 y un valor aproximado $\approx$ a éste.

n	4	5	6	7	8
$\|\boldsymbol{BF_n}\|$	2^{16}	2^{32}	2^{64}	2^{128}	2^{256}
$\approx$	6.10^4	4.10^9	10^{19}	10^{38}	10^{77}

Número de funciones Booleanas para determinado n.

Por tal motivo encontrar funciones Booleanas con propiedades específicas por medio de ensayo-error es computacionalmente imposible. Es por ello que se trabaja con clases de equivalencia de funciones. Hay también una explosión combinatoria importante pero más manejable para la cantidad de variables contempladas.

PREVENCIÓN DE ATAQUES

Como menciona Carlet en (Carlet, 2010 A) el diseño de sistemas criptográficos convencionales se basa en dos principios fundamentales introducidos por Shannon (Shannon, 1949): confusión y difusión:

- Confusión tiene por objeto ocultar cualquier estructura algebraica en el sistema y está fuertemente relacionado a la complejidad de las funciones Booleanas involucradas.
- Difusión tiene por objeto que, ante pequeños cambios en la entrada o las claves, tengamos grandes cambios en las salidas.

La resistencia de los criptosistemas a los ataques conocidos se puede cuantificar a través de algunas características fundamentales de las funciones Booleanas utilizados en ellos. El diseño de estas funciones criptográficas debe tener en cuenta varias características, o propiedades simultáneamente:

- Alto grado algebraico
- No linealidad
- Balanceada
- Resiliente
- Criterio de avalancha estricto
- Criterio de propagación

A continuación, describiremos cada una de éstas propiedades y sus definiciones.

GRADO ALGEBRAICO

Las funciones Booleanas utilizadas en cualquier criptosistema deben tener un grado algebraico alto ya que en caso contrario son propensas a ataques.

DEFINICION 1.2.I

Grado algebraico de una función *f*, es el número de entradas más grande que aparece en cualquier producto de la forma normal algebraica, definida en 2.1.II.

NO LINEALIDAD

Esta propiedad reduce el efecto de los ataques por criptoanálisis lineal.

DEFINICION 1.2.II

No linealidad de una función Booleana *f* es la mínima distancia de Hamming entre *f* y cualquier función afín $g(x) = mx + n$.

$$NL(f) = \min\{w(f \oplus g)\}$$

BALANCEADA

Las funciones Booleanas utilizadas deben ser balanceadas para evitar cualquier tipo de dependencia estadística entre la entrada y la salida de la función Booleana, lo cual puede ser utilizado en ataques a criptosistemas que no cumplan esta condición.

DEFINICION 1.2.III

Una función Booleana es balanceada si tiene la misma cantidad de 0's que de 1's en su tabla de verdad.

RESILIENCIA

Si la función Booleana no es resiliente entonces existe una correlación entre la salida de la función Booleana y sus entradas. Esta propiedad está relacionada con ataques a los generadores psuedo aleatorios usados en los criptosistemas.

DEFINICION 1.2.IV - RESILIENCIA

Una función Booleana es resiliente cuando tiene igual cantidad de 0's y 1's (balanceada) y además si se fija cualquier conjunto de k entradas entonces todas 2^k sub funciones tienen el mismo peso de Hamming (inmune a la correlación de orden k).

CRITERIO DE AVALANCHA ESTRICTO Y CRITERIO DE PROPAGACIÓN

Una función Booleana satisface el criterio de avalancha estricto si al complementar un solo bit de entrada resulta en un cambio en un solo bit de salida con una probabilidad de 1/2. Ambas propiedades están relacionadas con la difusión.

1.3 OBJETIVOS DE ESTE PROYECTO

El objetivo central del proyecto es la caracterización de las funciones Booleanas de n variables inmunes a la correlación de orden k de peso mínimo de Hamming para diversos valores de n y k según los rangos de la *Tabla 1 - Pesos mínimos de Carlet*.

OBJETIVOS A CUMPLIR EN EL PROYECTO

Dentro de los objetivos generales de la problemática planteada, el proyecto se enfocó en lograr:

1) Construir un algoritmo, de forma clara y eficiente que genere las clases de equivalencia correspondiente a funciones Booleanas de N variables, inmunes a la correlación de orden K y peso de Hamming mínimo (mayor a 0).

2) Construir otro algoritmo, que basado en la información resultante de la ejecución del primero, permita la generación aleatoria uniforme de funciones Booleanas inmunes a la correlación de menor peso de Hamming.

3) Brindar nueva información relacionada a las clases de equivalencia generadas: cantidades de funciones solución, características particulares de las clases de equivalencia solución, etc. Esta información permitirá analizar si es posible construir clases normales y enfocar el problema de otra manera en próximas investigaciones. Dado que el problema tiene muchas simetrías, entonces se pueden definir equivalencias entre clases. Una clase normal sería el representativo de estas clases.

4) Verificar los pesos mínimos presentados por Carlet (Carlet, 2010 A) y aportar valores aún desconocidos. A continuación, mostramos la tabla generada por Carlet que indica los pesos mínimos de Hamming de las funciones de *n* variables y orden *d* inmune a la correlación. Tanto el orden como la cantidad de variables varía entre 1 y 13.

n \ d	1	2	3	4	5	6	7	8	9	10	11	12	13
1	**2**												
2	**2**	**4**											
3	**2**	**4**	**8**										
4	**2**	**8**	**8**	**16**									
5	**2**	**8**	**16**	**16**	**32**								
6	**2**	**8**	**16**	**32**	**32**	**64**							
7	**2**	**8**	**16**	**64**	**64**	**64**	**128**						
8	**2**	**12**	**16**	**64**	**128**	**128**	**128**	**256**					
9	**2**	**12**	**24**	**128**	**128**	**256**	**256**	**256**	**512**				
10	**2**	**12**	**24**	**128**	**256**	*512*	*512*	*512*	*512*	*1024*			
11	**2**	**12**	**24**	?	?	*512*	*1024*	*1024*	*1024*	*1024*	*2048*		
12	**2**	**16**	**24**	?	?	?	*1024*	*2048*	*2048*	*2048*	*2048*	*4096*	
13	**2**	**16**	*32*	?	?	?	?	*4096*	*4096*	*4096*	*4096*	*4096*	*8192*

Tabla 1 - Pesos mínimos de Carlet

1.4 Resultados esperados

Los resultados esperados del proyecto están ligados a cada uno de los objetivos que se expusieron en el punto anterior:

- Algoritmo de generación de clases de equivalencia; se espera que el algoritmo de generación de clases de equivalencia, sea eficiente, correcto y que se puedan verificar cada una de las clases generadas por el mismo mediante comparación con resultados de otros autores o generación manual.

- Clases de equivalencia generadas; obtener la mayor cantidad de clases de equivalencia de peso mínimo.

- Tabla de Carlet; conseguir algún valor desconocido en dicha tabla y verificar los hallados.

1.5 Resumen de conclusiones

Se logró diseñar e implementar un algoritmo eficiente para el cálculo de clases inmunes a la correlación de orden k y n variables de peso de Hamming mínimo. Así como también la generación pseudo aleatoria de funciones Booleanas para determinado grado de inmune a la correlación y cantidad de variables.

Los resultados obtenidos fueron corroborados mediante la comparación con trabajos de otros autores, así como también con cálculos manuales realizados por nosotros.

En la ejecución de los algoritmos implementados se obtuvo información valiosa referida a la utilización de clases de equivalencia de funciones Booleanas. En particular al aumentar el valor de k se detectaron varias simetrías en la manera en que se dividían en clases las funciones que eran equivalentes para k-1. Estas simetrías permitirán agrupar las clases de equivalencia en equivalencia de segundo orden, definiendo clases normales representativas. Esto permite en trabajos futuros, re plantear la representación utilizada a clases normales con la cual se obtendrán mejores resultados al explotar estas simetrías.

CAPÍTULO 2 – MARCO TEÓRICO

2.1 CONCEPTOS BÁSICOS SOBRE FUNCIONES BOOLEANAS

A continuación, daremos algunos conceptos básicos para los cuales utilizaremos en su mayoría la notación empleada en "Equilivalence classes of Boolean functions for first-order correlation" de Jean-Marie y Alfredo Viola (Le Bars, y otros, 2010).

DEFINICIÓN 2.1.I – FUNCIÓN BOOLEANA

Una función Booleana es una función $f_n: \{0,1\}^n \rightarrow \{0,1\}$. Se puede representar de varias formas: por medio de tablas de verdad, de su forma normal algebraica, entradas con valor 1, transformada de Fourier, etc.

Una de las representaciones más utilizadas para una función Booleana es la forma explícita como una secuencia de 2^n 0's y 1's donde cada uno de esos valores corresponden al resultado de la función con *n* variables con determinada configuración. Por ejemplo, para una función Booleana con 3 variables podemos tener la siguiente configuración $f_3 = (10010110)$

También se puede definir la función Booleana mediante la tabla de verdad; esta tabla consta de 2^n filas, correspondientes a todas las combinaciones posibles de valores de variables y $n + 1$ columnas, donde la última columna es el valor que toma la función *f* para una configuración de valores de variables dada por la *n* columnas anteriores.

En el siguiente ejemplo daremos la tabla de verdad para la función $f_3 = (10010110)$

x_3	x_2	x_1	f_3
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

La representación por entradas en 1, indica qué entradas en la tabla de verdad tienen como resultado 1. En el caso de la función f_3 del ejemplo anterior serían: {(0,0,0);(0,1,1);(1,0,1);(1,1,0)}. Esta representación es mucho más compacta que expresar toda la función Booleana en su totalidad y puede ser útil cuando se procesan funciones de mucha cantidad de variables y un peso de Hamming pequeño.

DEFINICIÓN 2.1.II – FORMA NORMAL ALGEBRAICA

La forma normal algebraica es la representación de una función Booleana de n variables por medio un polinomio en n variables.

Para el ejemplo de f_3 se calcula de la siguiente manera:

A cada entrada de la tabla de verdad con resultado 1 se hace corresponder un polinomio: si $a_n a_{n-1} \dots a_1$ es la entrada de la tabla de verdad con resultado da 1, el polinomio correspondiente se forma de la siguiente manera:

$$a_i = 0 \longrightarrow 1 \oplus x_i$$
$$a_i = 1 \longrightarrow x_i$$

Para f_3 los polinomios serían:

x_3	x_2	x_1	f_3	Polinomio
0	0	0	1	$(1 \oplus x_3)(1 \oplus x_2)(1 \oplus x_1)$
0	0	1	0	
0	1	0	0	
0	1	1	1	$(1 \oplus x_3)x_2x_1$
1	0	0	0	
1	0	1	1	$x_3(1 \oplus x_2)x_1$
1	1	0	1	$x_3x_2(1 \oplus x_1)$
1	1	1	0	

Estos polinomios representan funciones que valen 1 cuando las variables en la fila de la tabla de verdad toman los valores allí indicados.

Por ejemplo, el polinomio $(1 \oplus x_3)(1 \oplus x_2)(1 \oplus x_1) = 1 \iff x_3 = x_2 = x_1 = 0$ con lo cual representa la función $\{1,0,0,0,0,0,0,0\}$.

Para obtener la forma normal algebraica de f_3 se hace el XOR de todas las entradas en 1:

$$f_3(x_3, x_2, x_1) = [(1 \oplus x_3)(1 \oplus x_2)(1 \oplus x_1)] \oplus [(1 \oplus x_3)x_2x_1] \oplus [x_3(1 \oplus x_2)x_1] \oplus [x_3x_2(1 \oplus x_1)]$$
$$f_3(x_3, x_2, x_1) = (1 \oplus x_3)[1 \oplus x_1 \oplus x_2 \oplus x_1x_2 \oplus x_1x_2] \oplus x_3[x_1 \oplus x_1x_2 \oplus x_2 \oplus x_1x_2]$$
$$f_3(x_3, x_2, x_1) = (1 \oplus x_3)[1 \oplus x_1 \oplus x_2] \oplus x_3[x_1 \oplus x_2]$$
$$f_3(x_3, x_2, x_1) = 1 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_1x_3 \oplus x_2x_3 \oplus x_3x_1 \oplus x_2x_3$$
$$f_3(x_3, x_2, x_1) = 1 \oplus x_1 \oplus x_2 \oplus x_3$$

La forma normal algebraica de la función Booleana f_3: $f_3(x_3, x_2, x_1) = 1 \oplus x_1 \oplus x_2 \oplus x_3$. Esta expresión vale 1 si y solo sí las x_i corresponden a una entrada de la tabla de verdad.

Como la forma algebraica de esta función tiene solo términos de grado 1, se dice que la función es afín y que tiene grado algebraico 1.

A continuación, daremos algunas definiciones, propiedades y teoremas sobre las funciones Booleanas que serán utilizadas a lo largo de este trabajo.

Definición 2.1.III - Peso de Hamming

El peso de Hamming de una función Booleana f es la cantidad de 1's que contiene su tabla de verdad.

Dada una función Booleana expresada de forma explícita como f_2= (1 1 1 0), el peso de Hamming de f_2 es 3. La representación que utilizaremos es $H_w(f_2) = 3$.

Lema 2.1.IV- Cota superior para el peso de Hamming

Dado $W(n,k) = \min\{H_w(f_n) | f_n \text{ es una función Booleana inmune a la correlación de orden } k\}$

Para cualquier n y k, el peso de Hamming de las funciones de n variables inmunes a la correlación de orden k

$$W(n,k) \leq 2W(n-1,k)$$

Demostración:

Esta desigualdad se debe a que concatenar dos funciones de n-1 variables inmunes a la correlación de orden k da una función inmune a la correlación de n variables de orden k (aunque no necesariamente de menor peso).

DEFINICIÓN 2.1.V - INMUNE A LA CORRELACIÓN DE ORDEN 1

Sea f_n una función Booleana con *n* variables y peso de Hamming *2m*. Entonces f_n es inmune a la correlación de orden 1 cuando:

$$H_w(f_n|x_i = 0) = H_w(f_n|x_i = 1) = m \; \forall \; i \in \{1, \dots, n\}$$

Denotamos $f_n|x_i = \varepsilon$ a la restricción de f_n condicionada a la valuación $x_i = \varepsilon$.

[Definición 3 de (Le Bars, y otros, 2010)]

DEFINICIÓN 2.1.VI - Inmune A LA CORRELACIÓN DE ORDEN K

Diremos que una función f_n es inmune a la correlación de orden k, con $k \leq n$ cuando fijando cualquier combinación de $i \leq k$ columnas de la tabla de verdad de f_n se obtiene el mismo peso de Hamming.

Daremos un ejemplo para facilitar la comprensión de la definición de inmune a la correlación, para el ejemplo utilizaremos la función f_3 empleada en el cálculo de forma normal algebraica.

Primero chequearemos si es inmune a la correlación de orden 1, utilizando la definición 2.1.V. Para esto fijamos en 0 y 1 una sola columna x_i con i= {1,2,3}.

con $x_1 = 0$ se obtiene peso 2 y con $x_1 = 1$ se obtiene peso 2, es decir que las sub tablas de verdad quedan de la siguiente manera:

$x_1 = 0$

x_3	x_2	f_3
0	0	1
0	1	0
1	0	0
1	1	1

$x_1 = 1$

x_3	x_2	f_3
0	0	0
0	1	1
1	0	1
1	1	0

con $x_2 = 0$ se obtiene peso 2 y con $x_2 = 1$ se obtiene peso 2, es decir que las sub tablas de verdad quedan de la siguiente manera:

$x_2 = 0$

x_3	x_1	f_3
0	0	1
0	1	0
1	0	0
1	1	1

$x_2 = 1$

x_3	x_1	f_3
0	0	0
0	1	1
1	0	1
1	1	0

con $x_3 = 0$ se obtiene peso 2 y con $x_3 = 1$ se obtiene peso 2, es decir que las sub tablas de verdad quedan de la siguiente manera:

$x_3 = 0$

x_2	x_1	f_3
0	0	1
0	1	0
1	0	0
1	1	1

$x_3 = 1$

x_2	x_1	f_3
0	0	0
0	1	1
1	0	1
1	1	0

Esto nos indica que f_3 es inmune a la correlación de orden 1.

De una forma similar es posible también probar que: $f_3$es inmune a la correlación de orden 2.

con $x_1 = 0$ y $x_2 = 0$ se obtiene peso 1 y con $x_1 = 1$ y $x_2 = 1$ se obtiene peso 1, es decir que las sub tablas de verdad quedan de la siguiente manera:

$x_1 = 0$ y $x_2 = 0$

x_3	f_3
0	1
1	0

$x_1 = 1$ y $x_2 = 1$

x_3	f_3
0	1
1	0

con $x_1 = 0$ y $x_3 = 0$ se obtiene peso 1 y con $x_1 = 1$ y $x_3 = 1$ se obtiene peso 1, es decir que las sub tablas de verdad quedan de la siguiente manera:

$x_1 = 0$ y $x_3 = 0$

x_2	f_3
0	1
1	0

$x_1 = 1$ y $x_3 = 1$

x_2	f_3
0	1
1	0

con $x_2 = 0$ y $x_3 = 0$ se obtiene peso 1 y con $x_2 = 1$ y $x_3 = 1$ se obtiene peso 1, es decir que las sub tablas de verdad quedan de la siguiente manera:

$x_2 = 0$ y $x_3 = 0$

x_1	f_3
0	1
1	0

$x_2 = 1$ y $x_3 = 1$

x_1	f_3
0	1
1	0

DEFINICIÓN 2.1.VII – FUNCIÓN BOOLEANA K RESILIENTE

Una función Booleana es k resiliente si además de ser balanceada (definición 1.2.III), es inmune a la correlación de orden *k*, con peso de Hamming 2^{n-1}.

DEFINICIÓN 2.1.VIII – CLASE DE EQUIVALENCIA

Una forma de agrupar funciones Booleanas con características similares es mediante **clases de equivalencia** que se representan como $< m, \delta_n, \dots, \delta_1 >$. Para funciones Booleanas inmunes a la correlación de orden 1: m es el peso de Hamming de la función y δ_n es la diferencia de pesos de Hamming cuando se fija $x_i = 0$ y $x_i = 1$.

Sea f_n una función Booleana con *n* variables. Para todo $i \in \{1, \dots, n\}$, definimos

$$\delta_i(f_n) = H_w(f_n|x_i = 0) - H_w(f_n|x_i = 1)$$

Luego tenemos que f_n es inmune a la correlación de orden 1 cuando $\delta_i(f_n) = 0$ para todo $i \in \{1, \dots, n\}$.

En el ejemplo de $f_3 = (1\ 0\ 0\ 1\ 0\ 1\ 1\ 0)$ la clase de equivalencia sería <4, 0, 0, 0>.

DEFINICIÓN 2.1.IX – COMPOSICIÓN DE FUNCIONES $*$

Toda función Booleana de n variables se puede generar con la ***composición (concatenación)*** de 2 funciones de n-1 variables:

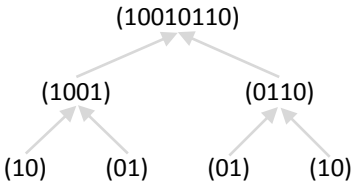
$$f_n = f_{n-1}^0 * f_{n-1}^1$$

Si aplicamos esta composición recursivamente, llegaremos a que toda función es composición de funciones de una variable. Esto se puede representar como un árbol binario. Además, esta misma composición es posible aplicarla a clases de equivalencia.

La función f_3 se puede descomponer en dos funciones de 2 variables fijando primero x_3 en 0 y luego x_3 en 1:

$f_3 = f_2^0 * f_2^1$ luego $f_2 = f_1^0 * f_1^1$ dando $f_3 = (f_1^{00} * f_1^{01}) * (f_1^{10} * f_1^{11})$

El árbol correspondiente es:



Notaciones:

Denotamos $\omega = \Omega(f_n) =< H_w(f_n), \delta_n(f_n), \dots, \delta_1(f_n) >$

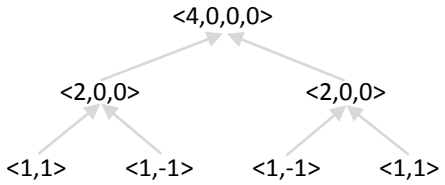
Además $\Omega_n^m = \{\omega \mid \exists\ f_n\ \omega = \Omega(f_n)\ y\ H_w(f_n) = m\ \}$

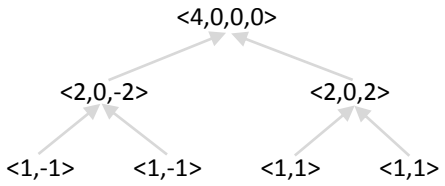
DEFINICIÓN 2.1.X – COMPOSICIÓN DE CLASE DE EQUIVALENCIA $*$

Sean $p, q \in \{0, \dots, 2^{n-1}\}$, $\omega^0 \epsilon\, \Omega_{n-1}^p$ y $\omega^1 \epsilon\, \Omega_{n-1}^q$.
El operador de clase $*$ lo definimos para $\omega^0 * \omega^1 = \omega$, donde $\omega \epsilon\, \Omega_n^m$ satisface el siguiente sistema:

$$\begin{cases} m = p + q \\ \delta_n = p - q \quad \forall\, i \in \{1, \dots, n-1\} \\ \delta_i = \delta_i^0 + \delta_i^1 \end{cases}$$

Como podemos ver a partir de la definición anterior, es posible construir las clases de equivalencia de forma recurrente a partir de dos clases de equivalencia más chicas, es decir clases que representan funciones con una variable menos.
Notar que la descomposición de una clase puede no ser única. A continuación, mostramos dos posibles árboles de composición para la clase de equivalencia <4, 0, 0, 0>.





COROLARIO 2.1.XI - CONTEO

Sea $\omega \epsilon\, \Omega_n$ entonces la cantidad de funciones de la clase ω es

$$|\omega| = \sum_{\omega^0 * \omega^1 = \omega} |\omega^0| * |\omega^1|$$

[Corolario 1 de (Le Bars, y otros, 2010)]

DEFINICIÓN 2.1.XII - CLASE ESPEJO

Dada la clase $\omega = \langle m, \delta_n, \dots, \delta_1 \rangle \in \Omega_n^m$, la clase espejo de ω es la clase

$$\overline{\omega} = \langle m, -\delta_n, \dots, -\delta_1 \rangle$$

[Definición 8 de (Le Bars, y otros, 2010)]

Para resolver nuestro problema es necesario obtener funciones Booleanas inmunes a la correlación de determinado orden, es fundamental a estudiar la ***transformada de Fourier*** y también la **transformada de Walsh**.

DEFINICIÓN 2.1.XIII - TRANSFORMADA DE FOURIER

Dada $f(x)$ una función Booleana y para cualquier $u \in \mathbb{F}_2^n$, la transformada de Fourier se define como:

$$\hat{f}(u) = \sum_{x \in \mathbb{F}_2^n} f\,(x)(-1)^{x \cdot u}$$

(Carlet, 2010 A)

DEFINICIÓN 2.1.XIV - TRANSFORMADA DE WALSH

Dada $f(x)$ una función Booleana y para cualquier $u \in \mathbb{F}_2^n$, la transformada de Walsh se define como:

$$\widehat{f_x}(\mathrm{u}) = \sum_{x \in \mathbb{F}_2^n} (-1)^{f(x) \oplus x \cdot u}$$

(Carlet, 2010 A)

DEFINICIÓN 2.1.XV – INMUNE A LA CORRELACIÓN DE ORDEN K EN BASE A LA TRANSFORMADA DE WALSH

Sea f_n una función Booleana con *n* variables y $k \in [1, m-1]$; la función f_n es inmune a la correlación de orden k si y solo sí su transformada de Walsh satisface:

$$\widehat{f_n}(u) = 0 \; \forall \; u \mid 1 \leq |u| \leq k$$

[Definición 2.1 de (P. Camion, et al., 1992)]

COROLARIO 2.1.XVI - CONSTRUCCIÓN RECURSIVA

$$\widehat{f_n}(vb) = \widehat{f_{n-1}^0}(v) + (-1)^b \widehat{f_{n-1}^1}(v), con\ b = 0{,}1$$

Demostración:

$\widehat{f_n}(vb) = \sum_{x \in F_2^n} (-1)^{f_n(x) + x.u}$ Si observamos que $x = y0$ o $x = y1$ con $y \in F_2^{n-1}$ tenemos que

$\widehat{f_n}(vb) = \sum_{y0} (-1)^{f_{n-1}^0(y) + (y.0).(vb)} + \sum_{y1} (-1)^{f_{n-1}^1(y) + (y.1).(vb)}$

$$\widehat{f_n}(vb) = \sum_{y} (-1)^{f_{n-1}^0(y) + yv} + \sum_{y} (-1)^{f_{n-1}^1(y) + yv + b} = \widehat{f_{n-1}^0}(v) + (-1)^b \widehat{f_{n-1}^1}(v)$$

DEFINICIÓN 2.1.XVII

Otra manera de definir una clase de equivalencia es la siguiente: $\omega = < H_w(f_n), \widehat{f_n}(u) \mid 1 \leq |u| \leq k >$

COROLARIO 2.1.XVIII

Sea $CI_{n,k}$ una clase de equivalencia que representa a las funciones Booleanas inmunes a la correlación de orden *k* y *n* variables. Entonces se cumple que:

$$CI_{n,k} = CI_{n-1,k}^0 * CI_{n-1,k}^1 \iff \begin{cases} CI_{n-1,k}^0 = < H_w(f_n), 0, \dots, 0, \widehat{f_n}(u) \mid |u| = k > \\ CI_{n-1,k}^1 = < H_w(f_n), 0, \dots, 0, -\widehat{f_n}(u) \mid |u| = k > \end{cases}$$

TEOREMA 2.1.XIX

Dado $W(n,k) = \min\{H_w(f_n) | f_n\ es\ una\ función\ Booleana\ inmune\ a\ la\ correlación\ de\ orden\ k\}$

Denotamos $w(n,k) = \frac{W(n,k)}{2^k}$

Dado n y k tal que $w(n,k) = 2^{n-k-1}$ se cumple que $w(n+p, k+p) = 2^{n-k-1}$ para cualquier $p \in \mathbb{N}$.

Demostración:

Basta con probar $w(n,k) = 2^{n-k-1} \Longrightarrow w(n+1, k+1) = 2^{n-k-1}$

Asumiendo que $w(n,k) = 2^{n-k-1}$ y dada f una función Booleana inmune a la correlación de orden k+1 y n+1 variables. Para cualquier $i \in \{1, \dots, n+1\}$ y cualquier $\varepsilon_i \in \mathbb{F}_2^n$, $f|_{x_i=\varepsilon_i}$ es una función inmune a la correlación de orden k y n variables. Por hipótesis, f es una función k-resiliente con n variables; esto implica que $H_w(f|_{x_i=\varepsilon_i}) = 2^{n-1}$.

Por lo tanto $H_w(f) = 2^n$, f es una función Booleana k+1 resiliente (inmune a la correlación de orden k+1 y balanceada).

2.2 CLASES NORMALES

Este problema presenta simetrías que son importantes explotar. Por ejemplo, en el caso k=1:

a) Si se renombran las variables, por ejemplo, se intercambian las variables x_i y x_j , las funciones Booleanas generadas son equivalentes en relación al problema. En este caso la clase $\langle m, \delta_n, \dots, \delta_i, \dots, \delta_j, \dots, \delta_1 \rangle$ se transforma en $\langle m, \delta_n, \dots, \delta_j, \dots, \delta_i, \dots, \delta_1 \rangle$.

Ejemplo: Si en la tabla de la izquierda intercambiamos x_3 por x_2 generamos la tabla de la derecha:

x_4	x_3	x_2	x_1	f_4
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Intercambiando x_3 por x_2 ⟷

y_4	y_3	y_2	y_1	f'_4
0	0	0	0	0
0	0	0	1	1
0	1	0	0	1
0	1	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	1	0	0	1
1	1	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	1	0	0
1	1	1	1	0

$$\omega_{izquierda} =< 7, 1, \mathbf{5}, \mathbf{-3}, -3 > \quad \omega_{derecha} =< 7, 1, \mathbf{-3}, \mathbf{5}, -3 >$$

b) Si se intercambian x_i de 0 a 1, las funciones generadas son equivalentes. La clase $\langle m, \delta_n, \dots, \delta_i, \dots, \delta_1 \rangle$ se transforma en $\langle m, \delta_n, \dots, -\delta_i, \dots, \delta_1 \rangle$.
Ejemplo: Si en la tabla de la izquierda intercambiamos x_4 de 0 a 1 generamos la tabla de la derecha:

x_4	x_3	x_2	x_1	f_4
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Intercambiando x_4 de 0 a 1 ⟷

x_4	x_3	x_2	x_1	f_4
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0

$$\omega_{izquierda} =< 7, \mathbf{1}, 5, -3, -3 > \quad \omega_{derecha} =< 7, \mathbf{-1}, -3, 5, -3 >$$

Entonces, a los efectos de capturar estas simetrías de manera algorítmica, se definen clases normales.

DEFINICIÓN 2.2.1 – CLASE NORMAL DE PRIMER ORDEN
Dado $m \leq 2^n$ y $\omega = \langle m, \delta_n, \dots, \delta_1 \rangle \in \Omega_n^m$, existe una permutación $\sigma : \{1, \dots, n\} \longrightarrow \{1, \dots, n\}$ que satisface:

$$\begin{cases} \alpha_i = \left|\delta_{\sigma(i)}\right| \\ \alpha_n \leq \alpha_{n-1} \leq \cdots \leq \alpha_1 \end{cases}$$

La clase $\theta = \langle m, \alpha_n, \dots, \alpha_1 \rangle$ es llamada clase normal de ω.

[Definición 17 de (Le Bars, y otros, 2010)]

Una clase $\langle m, \delta_n, \dots, \delta_1 \rangle$ es normal si se cumple que $0 \leq \alpha_n \leq \alpha_{n-1} \leq \cdots \leq \alpha_1$.
Además $\omega = \langle m, \delta_n, \dots, \delta_1 \rangle$ pertenece solamente a una clase normal N(ω).

Se puede ver que existen biyecciones entre clases de equivalencia, que se derivan de las biyecciones existentes entre funciones Booleanas, es decir si tomamos una función que pertenece a una clase ω podemos realizar ciertas permutaciones (Le Bars, y otros, 2010) en la función Booleana, manteniendo las mismas propiedades que cumple la función Booleana original, dando como resultado una función Booleana que pertenece a otra clase de equivalencia θ.

Por ejemplo, la clase normal N(<7,1,5,-3,-3>) = <7,1,3,3,5>.
011**10**00**10**11**1**0000 ∈ <7,1,5,-3,-3> y **1**11**01**00**01**11**0**0000 ∈ <7,1,3,3,5>

Podemos ver que las transformaciones que ocurren entre las clases de equivalencia son las siguientes:

- El valor 5 cambia de la posición 3 a la posición 5
- Los valores -3 se transforman en 3
- El valor -3 transformado en 3 cambia de la posición 4 a la posición 3
- El valor -3 transformado en 3 cambia de la posición 5 a la posición 4

2.3 ESTADO DEL ARTE

Se han formulado algunos algoritmos para la generación de clases de equivalencia correspondientes a funciones Booleanas inmune a la correlación, pero solo hasta orden 1 (Carrasco, et al., 2011). Estos trabajos también dan un gran aporte de definiciones y propiedades, así como también la demostración de teoremas y corolarios relacionados que han sido utilizados para la generalidad del problema.

Uno de éstos trabajos es el artículo de Le Bars y Viola llamado "Equivalence Classes of Boolean Functions for first-order correlation" (Le Bars, y otros, 2010) en el cual se describen y se dan propiedades de las funciones Booleanas inmunes a la correlación de primer orden y en particular las 1-resiliente (inmunes a la correlación y balanceadas). Algunas de estas propiedades se incluyen en el marco teórico de éste proyecto.

Por la trascendencia que tuvo en la solución al problema planteado, vale la pena destacar del trabajo de (Le Bars, y otros, 2010) la definición 8; ésta define un operador $*$ para construir una clase de n variables a partir de dos clases de n-1 variables. Esta definición es válida para la construcción recursiva de clases que representan funciones inmunes a la correlación de primer orden. Para nuestro trabajo fue necesario extender esta definición para poder aplicarla a cualquier orden k, el resultado de tal extensión es el Corolario 2.1.XVI del marco teórico.

En (Le Bars, y otros, 2010) también se presenta un método eficiente para el conteo de las funciones Booleanas que pertenecen a cierta clase de equivalencia, para esto define el Corolario 2.1.XI en el cual se calcula la cardinalidad de una clase de equivalencia utilizando la cardinalidad de las clases que la generan. Este Corolario fue utilizado en la solución de nuestro problema para poder contabilizar la cantidad de funciones inmunes a la correlación correspondientes a las clases de peso mínimo halladas para n variables y orden k.

Además de lo anterior, (Le Bars, y otros, 2010) también dan la definición 2.1.XI de clase espejo para los casos de funciones inmunes a la correlación de primer orden. En nuestro trabajo se extendió ésta definición para aplicarlo a todos los órdenes de correlación y poder optimizar la generación de clases de n variables a partir de clases de n-1 variables: al contarse con un listado de clases inmunes a la correlación de cierta cantidad n de variables, esta definición permitió en vez de realizar todas las combinaciones posibles (conjunto potencia de orden 2^n) solamente combinarlas con sus respectivas clases espejo (orden n).

Le Bars y Viola diseñaron algoritmos para el cálculo botton-up de clases que representan funciones inmunes a la correlación de primer orden y sus cardinalidades. La recorrida botton-up utilizada en estos algoritmos fue también empleada en el algoritmo solución al problema planteado en el proyecto.

Otro trabajo en el cual se estudian propiedades de las funciones Booleanas inmunes a la correlación es el realizado por Carrasco (Carrasco, et al., 2011); allí se plantea un algoritmo de generación ordenada de clases de equivalencia correspondientes a funciones Booleanas inmunes a la correlación.

Carrasco presenta una tabla en la cual se comparan los valores de cantidad de funciones, cantidad de clases de equivalencia y cantidad de clases normales para n entre 1 y 7; los valores allí expuestos alientan a la utilización en principio de las clases de equivalencia frente a funciones Booleanas. Esto fue lo realizado en nuestro trabajo; pero es notorio el lento crecimiento de la cantidad de clases normales con respecto a las clases de equivalencia para misma cantidad de variables.

n	# Funciones Booleanas	# Clases	# Clases normales
1	4	4	3
2	16	15	6
3	256	153	19
4	65536	5817	118
5	4294967296	936545	1755
6	18446744073709551616	632587361	73524
7	2^{128}	¿?	224949 (balanceadas)

Cantidad de funciones, clases y clases normales (Carrasco, et al., 2011)

Esta comparación lleva a plantear como un trabajo a futuro la adaptación de este proyecto para la utilización de clases normales, a los efectos de utilizar más eficientemente las simetrías del problema.

La forma de generar las funciones Booleanas presentada en el trabajo de Carrasco fue la base para la generación del algoritmo de generación aleatoria desarrollado en el proyecto.

Durante el tiempo en que fue desarrollado este proyecto, Jean-Marie Le Bars también se encontraba investigando la base teórica del problema y nos acercó a través del tutor nuevas propiedades avanzadas. Nos parece interesante citar algunas de estas propiedades que fueron utilizadas en el proyecto para el análisis de los datos obtenidos.

Dentro de las propiedades estudiadas y demostradas en los escritos de Jean Marie encontramos la prueba para el Corolario Cota superior peso de Hamming. Este corolario fue muy importante en el desarrollo de la solución ya que nos permitió identificar la cota inferior del peso a analizar en cada caso.

Otra de las demostraciones realizadas por Le Bars y utilizadas en el proyecto fue la demostración del Teorema sobre los pesos en las diagonales de la tabla de pesos mínimos. Este corolario permitió corroborar los pesos obtenidos por la solución elaborada en los casos que se cumplía la hipótesis del mismo.

Le Bars también nos envió otras propiedades relacionadas a las funciones Booleanas inmunes a la correlación; las mismas no fueron utilizadas para el análisis y desarrollo de la solución ya que la misma se enfocó en clases de equivalencia.

CAPÍTULO 3 – EL PROYECTO

3.1 DESCRIPCIÓN DEL PROYECTO

El desarrollo del proyecto constó de varias etapas, se relevó el estado del arte de los trabajos realizados sobre funciones Booleanas inmunes a la correlación y se complementó con el marco teórico relacionado al problema.

Luego de haber profundizado en ambos temas se diseñó un algoritmo que resolviera el problema, para ello se optó por representar las funciones Booleanas mediante clases de equivalencia. Por lo tanto, el objetivo del algoritmo diseñado es hallar las clases de equivalencia que representaran funciones Booleanas inmunes a la correlación de peso de Hamming mínimo.

Posteriormente al diseño de dicho algoritmo se definió la estructura de datos y la arquitectura de la aplicación que permitió tanto el cálculo de clases de equivalencia de peso mínimo para cierta cantidad de variables y un orden de correlación dado como la posibilidad de obtener aleatoriamente una función Booleana inmune a la correlación de cierto orden y una cantidad de variables dadas de peso mínimo.

Para poder almacenar las clases generadas, se utilizó una tabla de hash que se va a describir con mejor detalle en el Capítulo 4 y que básicamente tiene la siguiente estructura:

classes[n][k_extend][w][hash] = {class: counter}

Podemos observar que para cada clase de equivalencia (class), vamos a guardar la cantidad de funciones (counter) que pertenecen a la misma, además interesa saber datos particulares de la clase, tales como la cantidad de variables (n), el orden de correlación que se utiliza para generar la clase (k_corr), el peso de la clase (w) y un identificador (hash).

Después de tener definida la estructura, se seleccionó el lenguaje de programación en el cual se implementó el algoritmo. El mismo fue Python, ya que posee una manera eficiente de manejar la estructura de datos elegida, es sencillo integrar con distintos tipos de bases de datos y además los integrantes del equipo de desarrollo contaban con la experiencia suficiente como para utilizarlo de manera fluida.

Al finalizar la implementación del algoritmo en Python, se procedió a validar los resultados obtenidos en ambas funcionalidades de la aplicación y se concluyó con el análisis de los mismos.

3.2 VALIDACIÓN DE RESULTADOS

Para valores pequeños de cantidad de variables (n) y orden de correlación (k), la validación de los resultados obtenidos de nuevas clases, pesos mínimos y funciones correspondientes a determinada clase de equivalencia se realizó mediante comparación con cálculos manuales.

En el caso del peso mínimo la comparación de los resultados obtenidos se realizó con la tabla de Carlet, además de comprobar que el valor obtenido perteneciera al rango válido de peso para la cantidad de variables n:

$$H_w(n,k) \geq m \cdot 2^k \; con \; m \; \in \{1, \infty\}$$

La cantidad de funciones Booleanas correspondientes a cierta clase de equivalencia en el rango de variables n ∈ {1...5} se validó generando manualmente la clase y obteniendo las funciones correspondientes haciendo regresión en la generación de la misma.

Daremos un ejemplo de la generación manual de la clase para n=3 y k=2:

- Primero se calcularon todas las clases de peso mínimo para n=2 y k=1 extendidas hasta k=2. Del conjunto de clases resultado nos quedamos con las clases de peso mínimo mayor a 0, o sea las clases:

  <2|0, 0, -2> y <2|0, 0, 2>

- A partir de estas dos clases se genera la clase: <4|0, 0, 0| 0, 0, 0> de dos formas diferentes, siguiendo la construcción del Corolario 2:

  <4|0, 0, 0|0, 0, 0> = <2|0, 0, -2> ∗ <2|0, 0, 2>

  <4|0, 0, 0|0, 0, 0> = <2|0, 0, 2> ∗ <2|0, 0, -2>

- Luego para obtener las funciones correspondientes a la case obtenida para k=2: <4|0, 0, 0|0, 0, 0>, se hace regresión en el cálculo, pero tomando las clases sin extender.

  <2, 0, 0> = <1, -1> ∗ <1, 1>

  <2, 0, 0> = <1, 1> ∗ <1, -1>

  La clase <1, -1> es obtenida a partir de la función (0, 1) y la definición 2.1.X: $< 1, (0-1) >=< 1, -1 >$

  La clase <1, 1> es obtenida de igual modo a partir de la función (1, 0). Es por esto que la clase <2, 0, 0> representa a las funciones (0, 1, 1, 0) y (1, 0, 0, 1)

  Luego la clase de peso mínimo para n=3 y k=2 <4, 0, 0, 0, 0, 0, 0> representa a las funciones:

  (1001 0110)

  (0110 1001)

Con el cálculo realizado podremos verificar que la clase solución para n=3 k=2 es <4, 0, 0, 0, 0, 0, 0> con peso de Hamming mínimo igual a 4. Se verifica también que la cantidad de funciones correspondientes a dicha clase son 2.

En el Anexo II - Cálculos manuales de clases y funciones pueden verse los cálculos manuales realizados para valores de n ∈ {1...5} e inmune a la correlación de orden 2.

ALGUNOS DATOS OBTENIDOS

A continuación, listaremos todas las clases inmunes a la correlación de grado 1 y 2 (k) para 1, 2 y 3 variables (n), junto con el listado completo de las funciones que se corresponden a cada una de ellas.

Se mostrará también como se construyen dichas clases utilizando el operador $*$:

Clase k=1 n=1	Funciones correspondientes	Construcción de la clase
<0, 0>	(0 0)	Clase base calculada con def. 2.1.X
<2, 0>	(1 1)	Clase base calculada con def. 2.1.X

Clase k=1 n=2	Funciones correspondientes	Construcción de la clase
<0 , 0, 0>	(0 0 0 0)	<0, 0> * <0, 0>
<2, 0, 0>	(1 0 0 1)	<1, 1> * <1, -1>
<2, 0, 0>	(0 1 1 0)	<1, -1> * <1, 1>
<4, 0, 0>	(1 1 1 1)	<2, 0> * <2, 0>

Clase k=1 n=3	Funciones correspondientes	Construcción de la clase
<0, 0, 0, 0>	(0 0 0 0 0 0 0 0)	<0, 0, 0> * <0, 0, 0>
<2, 0, 0, 0>	(0 0 0 1 1 0 0 0)	<1, -1, -1> * <1, 1, 1>
<2, 0, 0, 0>	(0 1 0 0 0 0 1 0)	<1, 1, -1> * <1, -1, 1>
<2, 0, 0, 0>	(0 0 1 0 0 1 0 0)	<1, -1, 1> * <1, 1, -1>
<2, 0, 0, 0>	(1 0 0 0 0 0 0 1)	<1, 1, 1> * <1, -1, -1>
<4, 0, 0, 0>	(1 0 0 1 1 0 0 1)	<2, 0, 0> * <2, 0, 0>
<4, 0, 0, 0>	(1 1 0 0 0 0 1 1)	<2, 2, 0> * <2, -2, 0>
<4, 0, 0, 0>	(1 0 1 0 0 1 0 1)	<2, 0, 2> * <2, 0, -2>
<4, 0, 0, 0>	(0 0 1 1 1 1 0 0)	<2, -2, 0> * <2, 2, 0>
<4, 0, 0, 0>	(0 1 0 1 1 0 1 0)	<2, 0, -2> * <2, 0, 2>
<8, 0, 0, 0>	(1 1 1 1 1 1 1 1)	<4, 0, 0> * <4, 0, 0>

Clase k=2 n=2	Funciones correspondientes	Construcción de la clase
<0, 0, 0, 0>	(0 0 0 0)	<0, 0> * <0, 0>
<4, 0, 0, 0>	(1 1 1 1)	<2, 0> * <2, 0>

Clase k=2 n=3	Funciones correspondientes	Construcción de la clase
<0|0, 0, 0|0 , 0, 0>	(0 0 0 0 0 0 0 0)	<0|0, 0|0> * <0|0, 0|0>
<4|0, 0, 0|0, 0, 0>	(0 1 1 0 1 0 0 1)	<2|0, 0|-2> * <2|0, 0|2>
<4|0, 0, 0|0, 0, 0>	(1 0 0 1 0 1 1 0)	<2|0, 0| 2> * <2|0, 0|-2>
<8|0, 0, 0|0, 0, 0>	(1 1 1 1 1 1 1 1)	<4|0, 0| 0> * <4|0, 0|0>

Clase k=3 n=3	Funciones correspondientes	Construcción de la clase
<0|0, 0, 0|0, 0, 0>	(0 0 0 0 0 0 0 0)	<0|0, 0|0> * <0|0, 0|0>
<8|0, 0, 0|0, 0, 0>	(1 1 1 1 1 1 1 1)	<4|0, 0| 0> * <4|0, 0|0>

CAPÍTULO 4 – IMPLEMENTACIÓN

4.1 DESCRIPCIÓN DE LA IMPLEMENTACIÓN REALIZADA

Para poder cumplir con los objetivos del proyecto y llegar a los resultados esperados se analizó la realización de un programa en un lenguaje y repositorio de datos ágiles que permitiera la ejecución de algoritmos de búsqueda de clases y funciones Booleanas con ciertas características.

El programa que se elaboró cuenta con diferentes opciones de menú que entre ellas permite cumplir con los dos objetivos del proyecto: la búsqueda de clases de peso mínimo para cierta cantidad de variables y un orden de correlación dado; y la posibilidad de obtener aleatoriamente una función Booleana inmune a la correlación de cierto orden y una cantidad de variables dadas de peso mínimo.

En el Anexo I - Manual de la aplicación, se describen cada una de las opciones de menú implementadas, junto con ejemplos de ejecución.

4.2 TECNOLOGÍA/LENGUAJE/BD

Para la solución del problema, utilizamos el lenguaje Python en la versión 2.7 por las facilidades que brinda dicho lenguaje en el manejo de estructuras de datos como los diccionarios además de la integración con distintos tipos de bases de datos. Otro factor importante que fue tenido en cuenta en la elección del lenguaje fue la experiencia previa que teníamos en la utilización del lenguaje.

Con respecto al gestor de Base de datos en un principio se evaluó utilizar MySQL como motor de bases de datos, pero al ver las estructuras que manejamos (pocas relaciones entre ellas) y las cantidades de datos gigantescas, decidimos utilizar MongoDB el cual está pensado para esquemas no relacionales y manejo de grandes cantidades de datos. Este cambio lo implementamos luego de tener mucho desarrollado y el impacto fue grande en cuanto a la eficiencia.

4.3 HARDWARE

Para el desarrollo y las primeras pruebas de la solución del problema se utilizaron portables personales de mediano porte, con un procesador bueno y buena memoria RAM. A continuación, damos la descripción de uno de los equipos utilizados:

- Procesador: Intel(R) Core (™) i5 a 2.60 GHZ.
- Memoria RAM: 8GB.
- Disco Duro: 750GB.
- Sistema Operativo: Windows 8.1 de 64 bits.

Al avanzar en el desarrollo de la solución y requerir tiempos de ejecución más prolongados, así como también alta capacidad de memoria RAM se tomó la decisión de alquilar por dos meses un servidor dedicado. En este fue instalado el software de base necesario y se realizaron ejecuciones simples y en paralelo de la solución. El servidor alquilado contaba con las siguientes características:

- Procesador: AMD Opteron 3280
- Memoria RAM: 32GB.
- Disco Duro: 250GB SSD.
- Sistema Operativo: Ubuntu 14.04 de 64 bits.

4.4 IMPLEMENTACIÓN

Para la implementación de la solución, generamos un proyecto dividido en 3 módulos: uno que contiene el main principal (solution), otro que contiene las funciones utilitarias usadas (common) y otro para el manejo de persistencia en la base de datos (database).

ESTRUCTURAS DE DATOS UTILIZADAS

Las principales estructuras de datos manejadas son diccionarios, de los cuales vale la pena describir los siguientes:

- **diccionario classes**; utilizado para guardar todas las clases generadas a lo largo del programa y tiene la siguiente estructura:
    - *n (int)*, indica la cantidad de variables
    - *k_extend (int),* extensión de la clase
    - *w (int)*, peso de la clase
    - *hash (string)*, identificador de la clase generado haciendo un hash (sha 256) de los valores de la misma
    - *class (tuple of int)*, guarda la clase
    - *counter (int)*, indica cuántas funciones pertenecen a la clase

- **diccionario pedigree**; utilizado para guardar información sobre cómo se generó la clase, guardando para una determinada clase las referencias a las dos clases que le dieron origen. La estructura es la siguiente:
    - *n (int)*, indica la cantidad de variables
    - *k_extend (int),* extensión de la clase
    - *hash (string)*, identificador de la clase, generado haciendo un hash (sha 256) de los valores de la misma
    - *list of tuple(class_left, class_right)*, guarda una lista de referencia a las dos clases que dieron origen a la clase identificada por el hash

- **diccionario correlation**; utilizado para guardar todas las clases que son inmunes a la correlación para un cierto *n* y un cierto *k*. La estructura es la siguiente:
    - *n (int)*, indica la cantidad de variables
    - *k (int)*, indica el orden de correlación de la clase
    - *k_extend (int)*, extensión de la clase
    - *w (int)*, peso de la clase
    - *hash (string)*, identificador de la clase, generado haciendo un hash (sha 256) de los valores de la misma
    - *class (tuple of int)*, la clase
    - *counter (int)*, indica cuántas funciones pertenecen a la clase

- **diccionario solution**; utilizado para guardar todas las clases que son solución al problema, es decir, inmunes a la correlación de orden *k* con *n* variables y peso de Hamming (*w*) mínimo. La estructura es la siguiente:
    - *n (int)*, indica la cantidad de variables
    - *k (int)*, indica el orden de correlación de la clase
    - *class (tuple of int)*, guarda la clase
    - *counter (int)*, indica cuántas funciones pertenecen a la clase

- **diccionario partialclass**; utilizado para poder mejorar la búsqueda de clases espejo. Por ejemplo, para hallar la clase <4,0,0,0,0,0,0,0> (por ejemplo <2,0,0,-2> * <2,0,0,2>) que es una clase que representa a funciones de 3 variables, inmunes a la correlación de orden 2 y peso 4; lo que debemos hacer es obtener todas las clases que representan a funciones de 2 variables, inmunes a la correlación de orden 1 y peso 2 y cruzarlas para ver si el resultado genera la clase que queremos. Veamos más en detalle las clases de las funciones de 2 variables, por ejemplo, para <2,0,0,-2> vemos que el peso de Hamming es 2 (primera componente), la parte que corresponde a inmune a la correlación de orden 1 es (0,0) (segunda y tercera componente) y la parte que va a ayudar a construir una clase que represente a funciones de un n más, en este caso 3, es -2 (cuarta componente). Con lo cual en este diccionario vamos a guardar la información correspondiente a la cuarta componente que es la que nos va a interesar para espejar, es decir, vamos a buscar en este diccionario las que bajas ciertas condiciones, tengan la componente opuesta, en este caso las que tengan guardadas un 2. La estructura del diccionario es la siguiente:
    - *n (int)*, que indica la cantidad de variables
    - *k (int)*, indica el orden de correlación de la clase
    - *k_extend (int)*, extensión de la clase
    - *w (int)*, peso de la clase
    - *partial_class (tuple of int)*, parte de la tupla que interesa buscar el espejo
    - *ids (list of string)*, lista de identificadores de las clases que contiene la partial_class correspondiente al registro.

IMPLEMENTACIÓN DE BÚSQUEDA DE CLASES DE PESO MÍNIMO

A continuación, se muestra el pseudocódigo de funciones relevantes para la resolución de la primera parte del proyecto: la búsqueda de clases de peso mínimo para cierta cantidad de variables y un orden de correlación dado. Para cada una de ellas se dará una breve explicación de su funcionamiento.

FUNCIÓN RESOLVER

Esta función es empleada para la resolución de forma incremental e iterativa el cálculo de las clases de peso mínimo, comenzando por N = 2 y K = 1 hasta los valores pasados por parámetro.

Para cada caso de N y K, se consulta la solución inmediatamente anterior en diagonal, o sea N-1 K-1 para calcular el peso inicial por el que se buscarán soluciones.

En caso de haber solución para el caso anterior en diagonal, se extiende dicha solución empleándose la función *extender_solucion*. En caso contrario se procederá a calcular las clases inmune a la correlación para N-1, K-1 y peso 2^{k-1}.

Con el conjunto de clases generadas por extensión o calculadas, se procederá a calcular las nuevas clases correspondientes a N y K. Esto se realiza combinándolas una contra otra y generando la nueva clase. Se chequea que esta nueva clase sea inmune a la correlación de orden K y tenga peso múltiplo de 2^k; en caso de cumplirse estas condiciones es almacenada como solución. En caso de no encontrarse solución en esta iteración se aumentará el peso en 2^k y se volverá a realizar la búsqueda de clases solución.

```
Resolver(N, K)
        para n entre 2..N
                para k entre 1..K
                        peso = obtener_peso_sol(n - 1, k - 1)
                        mientras no encontramos solución
                                si ya tenemos una solución para n = n-1 y k = k-1
                                        set_correlations = extender_solucion(n-1, k-1)
                                si no
                                        set_correlations = GenerarCorr(n-1, peso/2, k-1)
```

```
                                para c0 en set_correlations:
                                        set_iter = buscar_espejos(correlation, n-1, k)
                                        para c1 en set_iter
                                                nueva_clase = c0 * c1
                                                si esCorrNK(nueva_clase, n, k)
                                                        //encontramos al menos una solucion
                                                        la agregamos y actualizamos a las
estructuras adecuadas (entre ellas el diccionario solution)
                                                fin de si
                                        fin de para
                                fin de para

                                si no encontramos solución
                                        peso = peso + 2^k
                                fin de si
                        fin de mientras
                        persistimos los datos
                fin de para //k
        fin de para //n
        desplegamos todos las soluciones obtenidas para los distintos n's y k's
fin de Resolver
```

FUNCIÓN EXTENDERSOLUCIÓN

Esta función recursiva es utilizada para generar la extensión de clases ya calculadas anteriormente para un orden (k_desde) a un orden superior (k_hasta). Para esto se utiliza la estructura de datos *pedigree* que permite obtener todas las clases que generaron esa clase y combinarlas para obtener una mayor extensión.

```
ExtenderSolucion(clase, n, k_desde, k_hasta)
        resultados = diccionario_vacio()
        pedigrees = cargar_de_bd(clase, n, k_desde)
        si n = 2 // paso base
             para pedigree en pedigress
                  c0, count0 = cargar_de_bd(pedigree.padre_izquierdo)
                  c1, count1 = cargar_de_bd(pedigree.padre_derecho)
                  clase_nueva = c0 * c1
                  resultados[clase_nueva] = obtener_contador(clase_nueva, count0, count1)
             retornar resultados

       //paso recursivo
        para pedigree en pedigrees
                elem_izq = ExtenderSolucion(pedigree.padre_izquierda, n-1, k_desde, k_hasta)

                elem_der = ExtenderSolucion(pedigree.padre_derecha  , n-1, k_desde, k_hasta)
                para cada c0, count0 en elementos_izquierda:
                        para cada c1, count1 en elementos_derecha:
                                clase_nueva = c0 * c1
                                //Actualizar estructuras de datos
                                resultados[clase_nueva] = obtener_contador(clase_nueva, count0, count1)
        retornar resultados
fin de ExtenderSolucion
```

FUNCIÓN BUSCARESPEJOS

Esta función permite buscar, dentro de las clases ya calculadas, las clases espejo que serán utilizadas para generar nuevas clases con la clase pasada por parámetro.

Para esto es fundamental el uso de la estructura de datos *partials.*

```
BuscarEspejos(clase, n, k)
        calcular el maximo k que se puede extender para ese n (k_extend)
        resultado = lista_vacia()
        peso = primera_componente(clase)
        tope = binomial(n, k)
        inicio = 1
        para i entre 1 y k-1
                inicio += binomial(n, i)
        componente_parcial = inverso(clase.obtener(inicio, inicio+tope))
        para cada id_clase en diccionario_partials(n, k, k_extend, peso, clase_parcial)
                si k > 1
                        clases = obtener_clases_diccionario_correlation(n, k-1, k_extend, peso, id_clase)
                        resultado.add(clases con sus contadores)
                si k = 1
                        clases = obtener_clases_diccionario_clases(n,k_extend,peso,id_clase)
                        resultado.add(clases con sus contadores)
        retornar resultado
fin de BuscarEspejos
```

FUNCIÓN GENERARCORR

La siguiente función se encarga de determinar el caso según el *k* pasado por parámetro e invocar a funciones que generan las clases inmunes a la correlación. Estas funciones guardan las clases generadas en la estructura *correlations* para luego poder ser retornadas como resultados de la función *GenerarCorr*.

```
GenerarCorr(n, peso, k)
        si k = 0
                GenerarTodasLasClases(n, peso)
                res = cargar_bd_clases(n, peso)
                retornar res
        si k = 1
                GenerarTodasLasClases(n-1, peso/2)
                GenerarKCorr(n, peso, k)
        si k > 1
                GenerarCorr(n-1, peso/2, k-1)
                GenerarKCorr(n, peso, k)
        cargar_bd_correlation(n, k, peso)
        si existe correlation para n, k y peso
                retornar correlation(n, k, peso)
        sino
                retornar lista vacía
fin de GenerarCorr
```

FUNCIÓN GENERARNUEVACLASE

Esta función es la encargada de generar las nuevas clases de *N* variables, extendidas hasta *k,* a partir de dos clases de *N-1* variables también extendidas hasta *k*.

```
GenerarNuevaClase(c0, c1, n, k) -> c0 * c1
si k > n
        k = n
suma = 0
prev_suma = 0
i = 0
nueva_clase = lista_vacia()
peso = primera_componente(c0) + primera_componente(c1)
nueva_clase[i] = peso
para i entre 1 y k
        tope = binomial(n-1, i-1)
        para j entre sum y sum+tope
                nueva_clase[i] = c0[prev_suma] - c1[prev_suma]
                prev_suma ++
                i++
        fin de para
        sum = sum + tope
        tope2 = binomial(n-1, i)
        para j entre sum y sum+tope2
                nueva_clase[i] = c0[prev_suma] + c1[prev_suma]
                prev_suma ++
                i ++
        fin de para
        sum = sum + tope2
        prev_suma = prev_suma - tope2
fin de para
retornar nueva_clase
fin de GenerarNuevaClase
```

FUNCIÓN ESCORRNK

Valida que la clase pasada como parámetro sea inmune a la correlación de orden *k* para *n* variables.

```
esCorrNK(clase, n, k)
        si primera_componente(clase) no es múltiplo de 2
                retornar Falso
        fin de si
        cant_ceros = 0
        para i entre 1 y k
                cant_ceros = cant_ceros + binomial(n, i)
        para k entre 1 y cant_ceros
                si clase[k] != 0
                        retornar Falso
                fin de si
        fin de para
        retornar Verdadero
fin de esCorrNK
```

FUNCIÓN GENERARTODASLASCLASES

Esta función es utilizada para generar todas las clases de peso igual al pasado por parámetro, independientemente de que sean inmunes a la correlación. Para esto se hallan las clases con todas las posibles combinaciones que formen el peso buscado.

```
GenerarTodasLasClases(n, peso)
        si existen las clases de n=n y peso=peso en la bd
                cargo las estructuras necesarias y retornamos
        sino
                para w entre 0 y peso
                   para cada clase c0 y count en ObtenerTodasLasClases(n-1, w)
                          para cada clase c1 y count1 en ObtenerTodasLasClases(n-1, peso-w)
                                  clase = c0 * c1
                                  actualizamos estructuras
                                  si esCorrNK(clase, n, 1)
agregamos clase a la estructura de corrs
                          fin de para
                   fin de para
                fin de para
        persistimos las estructuras
fin de GenerarTodasLasClases
```

FUNCIÓN GENERARKCORR

Es la función que calcula las clases inmunes a la correlación con los parámetros indicados. Además es invocada por la función anteriormente mencionada (GenerarCorr).

```
GenerarKCorr(n, peso, k)
        si k > 1
                clases = cargar_de_bd(correlations, n-1, k-1, peso/2)
        si no
                clases = cargar_de_bd(classes, n-1, peso/2)
        para cada clase c0 en clases
                para cada clase c1 en BuscarEspejos(c0, n-1, k)
                        clase = c0 * c1
si esCorrNK(clase, n, k)
        actualizamos estructuras
        fin de si
fin de para
        fin de para
fin de GenerarKCorr
```

IMPLEMENTACIÓN DE GENERACIÓN ALEATORIA DE FUNCIONES

En la primera parte se hallaron las clases de equivalencia correspondientes a funciones Booleanas inmunes a la correlación de orden k y n variables, cada una de estas clases representa un conjunto de funciones Booleanas. En esta segunda parte obtendrá una función en particular según un índice dado.

MÉTODO DE GENERACIÓN ALEATORIA

Para la generación aleatoria de funciones Booleanas de n variables e inmunes a la correlación de orden k es necesario primero ejecutar la búsqueda de clases inmunes a la correlación de peso mínimo para n y k. Esta búsqueda genera toda la información necesaria en la estructura de datos. Al finalizar la misma se estará en condiciones de ejecutar el algoritmo de generación aleatoria indicando el índice (i) de la función que se desea generar además de la clase solución correspondiente.

Este algoritmo fue desarrollado de modo que la función buscada, de n variables, se genere recursivamente buscando en las estructuras hasta llegar a los pasos bases correspondientes a funciones Booleanas de una variable.

Vale la pena mencionar que la forma de recorrer la estructura garantiza que cada vez que se ingresa un mismo índice para un n y k dados, la función Booleana generada es siempre la misma. Además, si se hace variar el índice entre 0 y la cantidad de funciones Booleanas menos uno (pertenecientes a la clase para n y k) se podrán obtener todas las funciones Booleanas pertenecientes a la clase.

A continuación, describiremos y daremos un ejemplo de ejecución de este algoritmo para los valores n=4, k=2 e i=3.

Primero se localiza la clase solución para el caso n y k; y se invoca al algoritmo pasando como paramento esta clase, n y el índice buscado.

El algoritmo busca la clase dentro de la estructura “solutions” y verifica que el índice i sea menor o igual a la cantidad de funciones Booleanas que pertenecen a la clase para n y k.

Para el ejemplo, los valores existentes en la estructura “solutions” son los siguientes:

Campo	Valor
n	4
k	2
class	<8,0,0,0,0,0,0,0,0,0,0>
counter	10

Estructura “solutions” para n=4 y k =2

Se puede ver que la clase solución <8,0,0,0,0,0,0,0,0,0,0> se corresponde con 10 funciones Booleanas, por lo que será posible generar la función correspondiente al índice pasado como parámetro (3 < 10).

Una vez validado que se puede generar la función Booleana correspondiente al índice indicado se consultan los datos generados en la estructura “pedigree”; esto nos permite saber de qué maneras se generó la clase solución y luego poder detectar la i-ésima generación que corresponderá a la función Booleana buscada.

Campo	Valor
N	4
k_extend	2
hash/class	<8,0,0,0,0,0,0,0,0,0,0>
list of tuple	[(<4, 0, 0, 0, 0, -4, 0>,1); (<4, 0, 0, 0, 0, 4, 0>,1)] [(<4, 0, 0, 0, 0, 0, 4>,1); (<4, 0, 0, 0, 0, 0, -4>,1)] [(<4, 0, 0, 0, -4, 0, 0>,1); (<4, 0, 0, 0, 4, 0, 0>,1)] [(<4, 0, 0, 0, 0, 4, 0>,1); (<4, 0, 0, 0, 0, -4, 0>,1)] ...

Estructura “pedigree” para n=4 y k =2

En la estructura pedigree, se almacena un listado de pares de clases (clase izquierda y clase derecha) que mediante la operación de concatenación dieron origen a la clase solución; cada una de estas clases cuenta con un contador de funciones Booleanas correspondientes.

Utilizando este contador, se va recorriendo el listado de pares de clases hasta detectar cuál se corresponde con la i-ésima función Booleana y al final de esta búsqueda se obtendrán los parámetros necesarios para invocar nuevamente a la generación recursiva, pero en este caso para la clase izquierda y la derecha.

Para el ejemplo, el par de clases que se corresponde al índice 3, es (<4, 0, 0, 0, 0, 4, 0>; <4, 0, 0, 0, 0, -4, 0>), los índices que serán utilizados en la invocación recursiva son 0 en ambos casos y el valor de n será 3.

La recursividad tiene un funcionamiento similar al descrito anteriormente y se realiza hasta llegar al paso base de n=1 en el cual se retorna la función correspondiente y se va concatenando hacia atrás.

Paso 0	**<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> n=4 i=3**							
Paso 1	<4, 0, 0, 0, 0, 4, 0> n=3 i=0				<4, 0, 0, 0, 0, -4, 0> n=3 i=0			
Paso 2	<2, 0, 2, 0> n=2 i=0		<2, 0, -2, 0> n=2 i=0		<2, 0, -2, 0> n=2 i=0		<2, 0, 2, 0> n=2 i=0	
Paso 3	<1, 1> n=1 i=0	<1, 1> n=1 i=0	<1, -1> n=1 i=0	<1, -1> n=1 i=0	<1, -1> n=1 i=0	<1, -1> n=1 i=0	<1, 1> n=1 i=0	<1, 1> n=1 i=0
Respuesta Paso 3	(1 0)	(1 0)	(0 1)	(0 1)	(0 1)	(0 1)	(1 0)	(1 0)
Respuesta Paso 2	(1 0 1 0)		(0 1 0 1)		(0 1 0 1)		(1 0 1 0)	
Respuesta Paso 1	(1 0 1 0 0 1 0 1)				(0 1 0 1 1 0 1 0)			
Respuesta Paso 0	(1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 0)							

A continuación, mostramos el pseudocódigo del algoritmo descripto anteriormente:

FUNCIÓN GETBOOLEANFUNCTION

Calcula de manera recursiva la función correspondiente a una clase e índice pasados por parámetro.

```
get_boolean_function(n, clase, indice):
        Si n == 1:
            retornar la funcion que se corresponde con esa clase
        contador = 0
        para cada pedigree de clase
            clase_izq, contador_izq = pedigree[0]
            clase_der, contador_der = pedigree[1]
            contador += contador izq*contador der
            si contador > indice:
                indice -= contador - contador_izq*contador_der
                break
        indice_izq = indice / contador_der
        indice_der = indice - (indice_izq * contador_derecha)
        retornar get_boolean_function(n-1, clase_izq, indice_izq) + get_boolean_function(n-1, clase_der,
indice_der)
```

En el Anexo I Manual de la aplicación, se puede visualizar un ejemplo de esta ejecución en “2->Get Boolean Functions”; también recomendamos mirar el ejemplo de generación de una clase mostrado en “31-> Print Tree of Correlation”.

CAPÍTULO 5 – RESULTADOS OBTENIDOS

5.1 ANÁLISIS DE RESULTADOS OBTENIDOS

El programa fue ejecutado en varios ciclos, brindando muchísima información muy valiosa que iremos analizando a lo largo de este capítulo y anexos.

Comenzaremos mostrando una tabla similar a la presentada por Carlet que ha sido llenada con los datos proporcionados por el programa. En el eje de las *y* se indica la cantidad de variables (n) mientras que en el eje de las *x* se indica el orden de correlación (k).

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	w 2 # 1												
2	w 2 # 2	w 4 # 1											
3	w 2 # 4	w 4 # 2	w 8 # 1										
4	w 2 # 8	w 8 # 10	w 16 # 2	w 16 # 1									
5	w 2 # 16	w 8 # 60	w 16 # 12	w 16 # 2	w 32 # 1								
6	w 2 # 32	w 8 # 240	w 16 # 60	w 32 # 14	w 32 # 2	w 64 # 1							
7	w 2 # 64	w 8 # 480	w 16 # 240	w 64 # 2186	w 64 16	w 64 # 2	w 128 # 1						
8	w 2 # 128	w 12 # 215040	w 16 # 480	w 64 # 21280	w 128 3770	w 128 18	w 128 # 2	w 256 # 1					
9	w 2 # 256	w 12	w 24 # 215040	w 128	w 128 # 21280	w 256 # 6092	w 256 # 20	w 256 # 2	w 512 # 1				
10	w 2 # 512	w 12	w 24	w 128	w 256	w 512	w 512 # 9352	w 512 # 22	w 512 # 2	w 1024 # 1			
11	w 2 # 1024	w 12	w 24	?	?	w 512	w 1024	w 1024 # 13774	w 1024 # 24	w 1024 # 2	w 2048 # 1		
12	w 2 # 2048	w 16	w 24	?	?	?	w 1024	w 2048	w 2048 # 19606	w 2048 # 26	w 2048 # 2	w 4096 # 1	
13	w 2 # 4096	w 16	w 32	?	?	?	?	w 4096	w 4096	w 4096 # 27120	w 4096 # 28	w 4096 # 2	w 8192 # 1

Tabla de peso mínimo y cantidad de funciones

Todos los valores en negro fueron hallados y comprobados por el programa elaborado, para cada valor de n y de k se muestran dos valores; el peso mínimo (w) correspondiente a las funciones de n variables inmune a la correlación de orden k y la cantidad de funciones encontradas (#).

Los valores en gris no pudieron ser hallados por el programa, mostramos el valor que Carlet indicaba en su tabla a modo de ilustrar cómo evoluciona el peso de estas clases.

Para cada combinación de cantidad de variables y orden de correlación se halló una única clase de equivalencia de peso mínimo; esta clase para algunas parejas de variable y orden se halla como la combinación de solo dos clases de n-1 variables, mientras que en otros casos es resultado de varias combinaciones diferentes.

Por un lado, se estudiaron los casos en que la clase era resultado de una única combinación y

BÚSQUEDA DE SIMETRÍAS

Sean f y g dos funciones Booleanas de n variables e inmunes a la correlación de orden k pertenecientes a una clase de equivalencia denotada $W_{n,k}$. Utilizando la transformada de Walsh, sabemos que en $W_{n,k}$ se cumple $\hat{f}(w) = \hat{g}(w)\ \forall\ 0 \leq |w| \leq k$, queremos ver qué pasa cuando $\hat{f}(w) = \hat{g}(w)\ \forall\ 0 \leq |w| \leq \boldsymbol{k+1}$, es decir a partir de $W_{n,k}$ ver qué pasa con $W_{n,k+1}$.

Al agregar |w|=k+1 la clase $W_{n,k}$ se particiona y esto nos hace cuestionarnos sobre lo que puede llegar a pasar, es decir, surgen preguntas del estilo: ¿cómo se particionan estas nuevas clases? ¿Se particionan en varias clases todas con la misma cardinalidad? ¿Se particionan en algunas clases con pocas funciones Booleanas asociadas y otras con muchas? ¿Hay alguna simetría en las funciones $\hat{f}(w)\ y\ \hat{g}(w)$ para |w| = k+1 que se pueda explotar ?

Es por esto que se realizaron extensiones de las clases a un orden más de correlación para ver la explosión combinatoria que daría como resultado.

En el Anexo IV se muestra el resultado de la realización de la extensión a 1 grado más sobre las clases ya halladas. Esto se realizó con el fin de estudiar posibles patrones en la extensión que permitieran en trabajos futuros trabajar con clases normales.

CAPÍTULO 6 - CONCLUSIONES

6.1 DIFICULTADES

Vale la pena mencionar que la mayor dificultad del problema a resolver se dio en la comprensión del mismo y no tanto en la implementación de la solución. El mismo tiene una fuerte componente matemática teórica, lo cual implicó estudiar y repasar propiedades fundamentales relativas a funciones Booleanas previo al comienzo de la elaboración de la solución.

Otra dificultad con la que nos encontramos al momento de diseñar la solución fue la elección tanto del lenguaje de programación como el software de base y hardware adecuados para la implementación y ejecución posterior de la solución. Para tomar esta decisión se tuvieron en cuenta muchos factores que comentamos en la sección: “Descripción de la implementación realizada”.

Luego de tener implementada la solución y que esta estuviera funcionando correctamente debíamos poder chequear los resultados que se estaban obteniendo. Para hacer esto se debió generar casos de prueba manualmente, lo cual generó cierta dificultad en el cálculo y esto fue solamente posible para casos de N {2...5} y orden de correlación K=2 por el gran crecimiento en el tamaño de las clases.

Atado a la dificultad anterior, al ejecutar la solución para valores mayores de N y k se vio un crecimiento exponencial de los cálculos efectuados y del volumen de datos generados. Para que la solución pudiera calcular más valores debimos realizar cambios que nos permitieron manejar mayor volumen de datos.

Entre los cambios realizados, se cambió de una Base de Datos relacional (MySQL) a una Base de Datos no relacional (Mongo DB). También realizamos mejoras en las estructuras de datos y optimizamos la solución para lograr realizar búsquedas más rápidas necesarias para el cálculo de nuevas clases.

Para poder avanzar aún más en el hallazgo de valores fue preciso mejorar el hardware sobre el cual corríamos la solución, otorgando más memoria y procesador al mismo. Esto fue posible alquilando un servidor muy potente y dedicado en el cual se dejó corriendo la solución varios días y permitió el hallazgo de algunos valores nuevos.

Lamentablemente no tuvimos tiempo para capturar simetrías definiendo clases normales que permitieran desarrollar algoritmos mucho más eficientes frente a la explosión combinatoria del problema al aumentar n y k.

6.2 OBJETIVOS VS RESULTADOS

Se elaboró una solución correcta: fue posible comprobar tanto contra cálculos manuales como contra la comparación con valores de bibliografía de base la correctitud de los valores hallados.

Bajo las condiciones de lenguaje, software de base y hardware utilizados para la ejecución de la solución se elaboró un algoritmo eficiente para la resolución del problema planteado. Una posible mejora en la eficiencia del algoritmo estaría dada por el cambio en el lenguaje de implementación del mismo; esto se plantea como trabajo de extensión a este proyecto.

Se logró obtener información adicional a la planteada como objetivo del proyecto; fue posible determinar la cantidad de funciones exactas asociadas a las clases de peso mínimo de n variables e inmunes a la correlación de orden k. También se logró obtener la información necesaria para poder retornar el listado completo de todas esas funciones.

Otra información adicional que es posible obtener utilizando los datos generados por la solución, es el árbol de generación de la misma. Todos estos datos pueden ser muy valiosos para trabajos de extensión relacionados a este proyecto.

Si bien se pudieron hallar y comprobar muchos de los pesos mínimos indicados por Carlet en su tabla, no fue posible el cálculo de nuevos valores desconocidos en dicha tabla. Consideramos que una mejora en la performance del algoritmo dada por una re-programación del mismo en otros lenguajes de más bajo nivel, la utilización de hardware con características similares o superiores al servidor utilizado, así como también incorporar la utilización de clases normales en el cálculo puede llevar a obtener valores desconocidos.

6.3 APORTES REALIZADOS

La realización de este proyecto permitió obtener varios resultados importantes en el campo de funciones Booleanas de peso mínimo e inmunes a la correlación de cierto orden. Dentro de los aportes más destacados tenemos:

- Generación de algoritmo para la obtención de funciones Booleanas a partir de clases inmunes a la correlación.

- Obtención de cantidad de funciones Booleanas correspondientes a las clases de peso mínimo calculadas.

- Análisis de resultados al extender clases en 1 grado, constatándose ciertos patrones que podrían ser objeto de un trabajo de extensión.

6.4 TRABAJOS A FUTURO

A medida que se fue desarrollando el proyecto, se fueron identificando ciertos aspectos que se podrían llegar a modificar para obtener más resultados o en menor tiempo. Consideramos que cada uno de estos aspectos puede llegar a formar parte de un trabajo de extensión del proyecto.

Nuestro trabajo estuvo basado en la utilización de clases de equivalencia de funciones Booleanas; otra forma de ver representado el problema podría ser mediante clases normales. No es trivial el uso de este tipo de clases, el algoritmo sería más complejo tanto en su diseño como implementación; pero el volumen de datos a manejar sería bastante menor. Un ejemplo de esto se puede ver en el trabajo realizado por Nicolás Carrasco para K=1 (Carrasco, et al., 2011).

Para poder enfocarlo desde esta otra perspectiva, una buena iniciativa podría ser inspeccionar los patrones que ocurren al extender las clases (ver Anexo IV) y tratar de agrupar las clases de equivalencia de acuerdo a estos patrones y así formar clases normales.

Otro enfoque que se pude dar al mismo problema es mediante la utilización de la representación de las funciones Booleanas mediante Orthogonals Arrays en el cual las funciones se representan mediante matrices y es posible establecer propiedades y teoremas para identificar funciones inmunes a la correlación. Esto llevaría nuestro problema a resolverse mediante cálculos sobre matrices. (Colbourn, et al., 2007)

Complementando el nuevo enfoque que planteamos, se debería acompañar de una implementación algorítmica basada en un lenguaje más eficiente que el utilizado en nuestro trabajo, como por ejemplo C. Esto permitiría tener un mejor manejo de la memoria y estructuras de datos a utilizar.

Además de lo anteriormente aconsejado, sería muy interesante poder diseñar el algoritmo para que ejecute ciertas operaciones de forma concurrente. Si bien esta mejora es llevada a cabo mediante cambios en el software se lograría un mayor resultado si se ejecuta el mismo en un cluster.

BIBLIOGRAFÍA

Carlet C. Boolean Functions for Cryptography and Error Correcting Codes [Publicación periódica] // Boolean Models and Methods in Mathematics, Computer Science, and Engineering. - 2010 A. - págs. 257-397.

Carlet C. Vectorial Boolean Functions for Cryptography. To appear in "Boolean Methods and Models" [Journal] // Cambridge University Press. - 2010 B.

Carlet Claude y Guilley Sylvain Correlation-immune Boolean functions and counter-measures to side channel attacks [Publicación periódica] // Applications of Algebra and Number Theory. - Linz : [s.n.], 2013.

Carrasco Nicolás, Le Bars Jean-Marie and Viola Alfredo Enumerative encoding of correlation-immune Boolean functions [Journal] // Information Theory Workshop (ITW), 2011 IEEE. - 2011. - pp. 643 - 647.

Clifford W.K. On the type of compound statement involving fours classes [Journal] // Mem. Lit. Philosophic Society. - 1877. - pp. 88–101.

Colbourn Charles J. and Dinitz Jeffrey H. Handbook of Combinatorial Designs [Book]. - [s.l.] : Chapman & Hall/CRC, 2007.

E.M. Palmer R.C. Read, R.W. Robinson Balancing the n-cube: a census of colorings [Journal] // Journal Algebraic Combinatorics 1. - 1992. - pp. 227–241.

Henríquez Francisco Rodríguez De la búsqueda de funciones booleanas con buenas propiedades criptográficas [Publicación periódica] // CINVESTAV. - 2007. - págs. 50-65.

Jevons W.S. The Principles of Science [Book]. - London : Macmillan, 1877.

Le Bars Jean Marie y Viola Alfredo Equilivalence classes of Boolean functions for first-order correlation [Publicación periódica] // IEEE Transactions on Information Theory. - 2010. - págs. 1247–1261.

Massey J.L. Shifttregister synthesis and BCH decoding [Journal] // IEEE Transactions on Information Theory. - 1969. - pp. 22-127.

NIST DATA ENCRYPTION STANDARD (DES) [Conference]. - [s.l.] : U.S. DEPARTMENT OF COMMERCE, 1999. - p. 17.

P. Camion [et al.] On correlation-immune functions [Journal] // Advances in Cryptology: Crypto '91. - 1992. - pp. 86-100.

Shannon C.E. Communication theory of secrecy systems [Article] // Bell system technical journal. - 1949. - pp. 656-715.

Siegenthaler T. Correlation-immunity of linear combining functions for cryptographic applications [Journal] // IEEE Transactions on Information Theory. - 1984. - pp. 776-780.

Stinson Douglas R. Cryptography Theory and practice [Book]. - Ontario, Canada : Chapman & Hall/CRC, 2006.

Strazdins I. Universal affine classification of boolean functions [Journal] // Acta Applicandae Mathematicae 46. - 1997. - pp. 147–167.

Technology National Institute of Standards and Data Encryption Standard (DES) [Journal] // FIPS PUB 46-3. - 1999. - p. 17.

ANEXO I - MANUAL DE LA APLICACIÓN

I.1 REQUERIMIENTOS

Para poder ejecutar la aplicación es necesario que se instalen los siguientes programas:

- Pyhton 2.7
- MongoDB 2.6 Standard
- Plugin pymongo 2.7

I.2 DETALLE DE LA APLICACIÓN

La aplicación generará al iniciarse una Base de Datos en Mongo con el nombre “minimumweight”; en dicha base se almacenarán todos los datos calculados, entre ellos las clases de equivalencia buscadas.

El menú de la aplicación cuenta con una gran gama de opciones que permiten calcular las clases de equivalencia soluciones y obtener las funciones Booleanas correspondientes:

```
1 -> Calculate All Minimum Weights
11-> Calculate Individual Minimum Weight
12-> Calculate Classes
13-> Calculate Correlation
2-> Get Boolean Function
22-> Get All Boolean Functions
3-> Print Solutions
31-> Print Tree of Correlation
32-> Calculate Extended Classes
400-> Drop Database
5-> Exit
```

A continuación, detallaremos cada una de las opciones indicando que entradas requiere y que salidas produce:

1 -> CALCULATE ALL MINIMUM WEIGHTS

Para una cantidad N de variables y un orden de correlación K ingresados como parámetros se calculan las clases solución de N={2..N} y K={2..K}. Se invoca a la función: solve()

Al finalizar el cálculo se imprime la cantidad de tiempo insumida en segundos.

Ejemplo de ejecución:

```
Select option: 1
Select N: 3
Select K: 3
(n, k) = (3,3)
>>>> Weight: 2 <-> N = 2, K = 1
All 0 corr of weight 1 VS all of weight 1 (N=1)
[1/2] ---> 0-corr, w=1, n=1, Size_Partial=39
All 0 corr of weight 1 VS all of weight 1 - OK (N=1)
Minimum Weight Solutions
N = 2, K = 1, W = 2, #Functions = 2 => {(2, 0, 0): 2}
```

```
EAT: 0.0529999732971 seconds
>>>> Weight: 4 <-> N = 2, K = 2
All 1 corr of weight 2 VS all of weight 2 (N=1)
[1/1] ---> 1-corr, w=2, n=1, Size_Partial=39
All 1 corr of weight 2 VS all of weight 2 - OK (N=1)
Minimum Weight Solutions
N = 2, K = 2, W = 4, #Functions = 1 => {(4, 0, 0, 0): 1}
EAT: 0.105999946594 seconds
>>>> Weight: 2 <-> N = 3, K = 1
Calculating for N = 2 and weight = 1
Crossing All Classes of weight 0 VS of weight 1 --> N = 1
Crossing All Classes of weight 1 VS of weight 0 --> N = 1
END :: Crossing All Classes of weight 1 (N=1)
All 0 corr of weight 1 VS all of weight 1 (N=2)
[1/4] ---> 0-corr, w=1, n=2, Size_Partial=39
All 0 corr of weight 1 VS all of weight 1 - OK (N=2)
Minimum Weight Solutions
N = 3, K = 1, W = 2, #Functions = 4 => {(2, 0, 0, 0): 4}
EAT: 0.27799987793 seconds
>>>> Weight: 4 <-> N = 3, K = 2
All 1 corr of weight 2 VS all of weight 2 (N=2)
[1/2] ---> 1-corr, w=2, n=2, Size_Partial=39
All 1 corr of weight 2 VS all of weight 2 - OK (N=2)
Minimum Weight Solutions
N = 3, K = 2, W = 4, #Functions = 2 => {(4, 0, 0, 0, 0, 0, 0): 2}
EAT: 0.391999959946 seconds
>>>> Weight: 8 <-> N = 3, K = 3
All 2 corr of weight 4 VS all of weight 4 (N=2)
[1/1] ---> 2-corr, w=4, n=2, Size_Partial=39
All 2 corr of weight 4 VS all of weight 4 - OK (N=2)
Minimum Weight Solutions
N = 3, K = 3, W = 8, #Functions = 1 => {(8, 0, 0, 0, 0, 0, 0, 0): 1}
EAT: 0.464999914169 seconds
```

11-> CALCULATE INDIVIDUAL MINIMUM WEIGHT

Para una cantidad N de variables y un orden de correlación K ingresados como parámetros se calculan las clases de equivalencia solución que contienen funciones Booleanas de N variables y orden de correlación K.

Se invoca la misma función que en la opción 1 pero indicando que el cálculo es particular para los valores ingresados. solve(True)

Al finalizar el cálculo se imprime la cantidad de tiempo insumido en segundos.

Ejemplo de ejecución:

```
Select option: 11
Select N: 5
Select K: 3
(n, k) = (5,3)
>>>> Weight: 8 <-> N = 5, K = 3
```

```
Calculating for N = 2 and weight = 1
Crossing All Classes of weight 0 VS of weight 1 --> N = 1
Crossing All Classes of weight 1 VS of weight 0 --> N = 1
END :: Crossing All Classes of weight 1 (N=1)
Generating all 1-corr of weight exactly = 2 (N=3)
[1/4] ---> 1-corr, w=2, n=3, Size_Partial=39
END:: Generating all 1-corr of weight exactly = 2 (N=3)
Generating all 2-corr of weight exactly = 4 (N=4)
[1/4] ---> 2-corr, w=4, n=4, Size_Partial=39
END:: Generating all 2-corr of weight exactly = 4 (N=4)
All 2 corr of weight 4 VS all of weight 4 (N=4)
All 2 corr of weight 4 VS all of weight 4 - OK (N=4)
>>>> Weight: 16 <-> N = 5, K = 3
Calculating for N = 2 and weight = 2
Crossing All Classes of weight 0 VS of weight 2 --> N = 1
Crossing All Classes of weight 1 VS of weight 1 --> N = 1
Crossing All Classes of weight 2 VS of weight 0 --> N = 1
END :: Crossing All Classes of weight 2 (N=1)
Generating all 1-corr of weight exactly = 4 (N=3)
[1/6] ---> 1-corr, w=4, n=3, Size_Partial=39
END:: Generating all 1-corr of weight exactly = 4 (N=3)
Generating all 2-corr of weight exactly = 8 (N=4)
[1/8] ---> 2-corr, w=8, n=4, Size_Partial=39
END:: Generating all 2-corr of weight exactly = 8 (N=4)
All 2 corr of weight 8 VS all of weight 8 (N=4)
[1/9] ---> 2-corr, w=8, n=4, Size_Partial=39
All 2 corr of weight 8 VS all of weight 8 - OK (N=4)
Minimum Weight Solutions
N = 5, K = 3, W = 16, #Functions = 12 => {(16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0): 12}
EAT: 0.682999849319 seconds
```

12-> CALCULATE CLASSES

Esta opción calcula todas las clases de equivalencia que contienen funciones Booleanas de N variables extendidas hasta un K dado y de peso W.

Se invoca la función: calculate_classes(n, k, w)

Como resultado se imprime el listado de clases de equivalencia con la cantidad de funciones Booleanas correspondientes y el resumen de la cantidad de clases y funciones calculadas.

Ejemplo de ejecución:

```
Select option: 12
Select N: 2
Select K_extend: 2
Select W: 2
(n, k_extend, w) = (2,2,2)
#: 1 - class: (2, -2, 0, 0)
#: 1 - class: (2, 0, 2, 0)
```

```
#: 1 - class: (2, 0, -2, 0)
#: 1 - class: (2, 2, 0, 0)
#: 1 - class: (2, 0, 0, -2)
#: 1 - class: (2, 0, 0, 2)
Size of classes: 6
Size of functions: 6
```

13-> CALCULATE CORRELATION

Esta opción calcula las clases de equivalencia correspondientes a funciones Booleanas de N variables, inmunes a la correlación de orden K, peso W y extendidas hasta K_extend.

Al finalizar el cálculo se imprime el listado de clases de equivalencia con la cantidad de funciones Booleanas correspondientes y un resumen de cantidad de clases y funciones.

Se invoca a la función: calculate_correlation(n, k_extend, k, w)

```
Select option: 13
Select N: 3
Select K: 2
Select K_extend: 2
Select W: 4
(n, k, k_extend, w) = (3,2,2,4)
Generating all 1-corr of weight exactly = 2 (N=2)
[1/2] ---> 1-corr, w=2, n=2, Size_Partial=39
END:: Generating all 1-corr of weight exactly = 2 (N=2)
Generating all 2-corr of weight exactly = 4 (N=3)
[1/2] ---> 2-corr, w=4, n=3, Size_Partial=39
END:: Generating all 2-corr of weight exactly = 4 (N=3)
#: 2 - correlation: (4, 0, 0, 0, 0, 0, 0)
Size of correlations: 1
Size of functions: 2
```

2-> GET BOOLEAN FUNCTION

Esta opción devuelve la función Booleana de índice I dentro del conjunto de soluciones correspondientes a funciones Booleanas de N variables, extendidas hasta K, orden de correlación K e índice dentro del conjunto de soluciones. El índice comienza en 0.

Ejemplo de ejecución:

```
Select option: 2
Select N: 4
Select K: 2
Select I: 5
(n, k, i) = (4,2,5)
0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1
```

22-> GET ALL BOOLEAN FUNCTIONS

Esta opción devuelve todas las funciones Booleanas para N variables, extendidas hasta K y orden de correlación K.

Ejemplo de ejecución:

```
Select option: 22
Select N: 4
Select K: 2
(n, k) = (4,2)
0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1
1 0 0 1 1 0 0 1 0 1 1 0 0 1 1 0
0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 1
1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 0
1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0
0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1
1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0
1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1
0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0
0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1
Size of result 10
```

3-> PRINT SOLUTIONS

Imprime la solución del cálculo realizado previamente correspondiente a N variables y orden de correlación K.

Invoca a la función load_solution para cargar los datos previamente calculados y luego invoca a la función print_results para imprimir el resultado.

Ejemplo de ejecución:

```
Select option: 3
Select N: 4
Select K: 3
(n, k) = (4,3)
Minimum Weight Solutions
N = 4, K = 3, W = 8, #Functions = 2 => {(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0): 2}
```

31-> PRINT TREE OF CORRELATION

Imprime el árbol de generación para una clase de equivalencia dada. Los parámetros de entrada son: la cantidad de variables (N) de la clase, el orden de correlación (K), el índice de extensión (k_extend), el peso (W) de la clase y la clase en sí (Class). La salida es la impresión de los antecesores de la clase, o sea a partir de que otras clases se generó.

Se invoca a la función: calculate_tree(n,k_extend,k,w, class_tree)

Ejemplo de ejecución:

```
Select option: 31
Select N: 4
Select K: 4
Select K_extend: 4
Select W: 16
Class (-1 no class): 16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
(n, k, k_extend, w) = (4,4,4,16)

Class: (16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
(8, 0, 0, 0, 0, 0, 0, 0) * (8, 0, 0, 0, 0, 0, 0, 0)
(8, 0, 0, 0, 0, 0, 0, 0)
(4, 0, 0, 0) * (4, 0, 0, 0)
(4, 0, 0, 0)
(2, 0) * (2, 0)
(2, 0)
(2, 0)
(4, 0, 0, 0)
(2, 0) * (2, 0)
(2, 0)
(2, 0)
(8, 0, 0, 0, 0, 0, 0, 0)
(4, 0, 0, 0) * (4, 0, 0, 0)
(4, 0, 0, 0)
(2, 0) * (2, 0)
(2, 0)
(2, 0)
(4, 0, 0, 0)
(2, 0) * (2, 0)
(2, 0)
(2, 0)
```

32-> CALCULATE EXTENDED CLASSES

Esta opción calcula para una clase en particular su extensión. Los parámetros de entrada son: clase a extender, N correspondiente a la cantidad de variables, orden de correlación de la clase a extender y el nuevo índice de extensión.

Se invoca a la función: calculate_extended_classes(class, N, extend_from, extend_to)

Ejemplo de ejecución:

```
Select option: 32
Class to extend: 8,0,0,0,0,0,0,0,0,0,0
Select N: 4
Select original extended k: 2
Select K to extend: 3

Original class: (8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

Extending the class from k=2 to k=3
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -8): 1
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -8, 0, 0, 0): 1
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0): 2
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0): 1
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8): 1
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -8, 0): 1
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0): 1
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 0, 0): 1
(8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -8, 0, 0): 1
Total: 9 classes
```

400-> DROP DATABASE

Borra por completo el contenido de la base de datos generada por la aplicación.

5-> EXIT

Finaliza la ejecución de la aplicación

ANEXO II - CÁLCULOS MANUALES DE CLASES Y FUNCIONES

En este anexo presentamos todos los cálculos hechos manualmente para valores de N ∈ {2..5} y orden de correlación K=2. Estos cálculos nos fueron útiles para la validación de los resultados obtenidos por la aplicación en estos rangos de valores.

Para poder generar clases de peso mínimo P, inmunes a la correlación de orden K y N variables, es necesario realizar todas las combinaciones posibles de clases con las siguientes características:

- mitad del peso (por LEMA 2.1.IV - COTA SUPERIOR PARA EL PESO DE HAMMING)
- una variable menos (N-1)
- inmune a la correlación de un orden menos (K-1)
- extendidas hasta K: calculados todos los valores de Fourier de peso de Hamming K

De todas estas combinaciones se filtran únicamente las clases de la forma <P,X0's> donde P es el peso mínimo. Esta forma representa clases inmunes a la correlación de orden K.

En el caso de **N2 K2,** se tomaron las clases de peso 2, inmunes a la correlación de orden 1 y 1 variable. Estas últimas corresponden únicamente a la clase: <2,0> que es originada por la función (1 1).

Combinando esta clase consigo misma se obtiene la clase <4,0,0,0> con una única función asociada a la combinación: (1111)

En resumen, la clase <4,0,0,0> = <2,0>*<2,0>; función: (1 1 1 1)

A continuación, mostraremos únicamente la generación y funciones asociadas correspondientes a los cálculos para N3 K2; N4 k2 y N5 k2

N3 K2

```
<4, 0, 0, 0, 0, 0, 0> = <2, 0, 0, -2> * <2, 0, 0, 2>; función: (0110 1001)
<4, 0, 0, 0, 0, 0, 0> = <2, 0, 0, 2> * <2, 0, 0, -2>; función: (1001 0110)

Referencias
<2, 0, 0, -2> = <1,-1> * <1,1> función (0110)
<2, 0, 0, 2> = <1,1> * <1,-1> función (1001)
<1,-1> función (01)
<1,1> función (10)
```

N4 K2

```
<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 4, 0> * <4, 0, 0, 0, 0, -4, 0> : 1 función
  (1010 0101 0101 1010)
<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0> * <4, 0, 0, 0, 0, 0, 0> : 4 funciones
   (0110 1001 0110 1001)
   (0110 1001 1001 0110)
   (1001 0110 1001 0110)
   (1001 0110 0110 1001)
<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, -4, 0, 0> * <4, 0, 0, 0, 4, 0, 0> : 1 función
   (0011 1100 1100 0011)
<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, -4, 0> * <4, 0, 0, 0, 0, 4, 0> : 1 función
   (0101 1010 1010 0101)
```

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 4> * <4, 0, 0, 0, 0, 0, -4> : 1 función
(1001 1001 0110 0110)
<8, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, -4> * <4, 0, 0, 0, 0, 0, 4> : 1 función
(0110 0110 1001 1001)
<8, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 4, 0, 0> * <4, 0, 0, 0, -4, 0, 0> : 1 función
(1100 0011 0011 1100)

Referencias
<4, 0, 0, 0, 0, -4, 0> = <2, 0, -2, 0> * <2, 0, 2, 0>
<4, 0, 0, 0, 0, 0, 4> = <2, 0, 0, 2> * <2, 0, 0, 2>
<4, 0, 0, 0, 0, 0, 0> = <2, 0, 0, 2> * <2, 0, 0, -2>
<4, 0, 0, 0, -4, 0, 0> = <2, -2, 0, 0> * <2, 2, 0, 0>
<4, 0, 0, 0, 0, 0, 0> = <2, 0, 0, -2> * <2, 0, 0, 2>
<4, 0, 0, 0, 0, 0, -4> = <2, 0, 0, -2> * <2, 0, 0, -2>
<4, 0, 0, 0, 0, 4, 0> = <2, 0, 2, 0> * <2, 0, -2, 0>
<4, 0, 0, 0, 4, 0, 0> = <2, 2, 0, 0> * <2, -2, 0, 0>

N5 K2

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 4, 0, 0, -4, 0> * <4, 0, 0, 0, 0, 0, -4, 0, 0, 4, 0> : 1 función

<4, 0, 0, 0, 0, 0, 4, 0, 0, -4, 0> = <2, 0, 2, 0, 0, -2, 0> * <2, 0, -2, 0, 0, -2, 0> : 0100 1000 0001 0010

<2, 0, 2, 0, 0, -2, 0> = <1, 1, -1, -1> * <1, 1, 1, 1> : 0100 1000

<2, 0, -2, 0, 0, -2, 0> = <1, -1, -1, 1> * <1, -1, 1, -1> : 0001 0010

<4, 0, 0, 0, 0, 0, -4, 0, 0, 4, 0> = <2, 0, -2, 0, 0, 2, 0> * <2, 0, 2, 0, 0, 2, 0> : 0010 0001 1000 0100

<2, 0, -2, 0, 0, 2, 0> = <1, -1, 1, -1> * <1, -1, -1, 1> : 0010 0001

<2, 0, 2, 0, 0, 2, 0> = <1, 1, 1, 1> * <1, 1, -1, -1> : 1000 0100

===> (0100 1000 0001 0010 0010 0001 1000 0100)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0> * <4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 0> : 4 funciones

<4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0> = <2, 2, 0, 0, 0, 0, 2> * <2, -2, 0, 0, 0, 0, -2> : 1001 0000 0000 0110

<2, 2, 0, 0, 0, 0, 2> = <2, 0, 0, 2> * <0, 0, 0, 0> : 1001 0000

<2, -2, 0, 0, 0, 0, -2> = <0, 0, 0, 0> * <2, 0, 0, -2> : 0000 0110

<4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0> = <2, 2, 0, 0, 0, 0, -2> * <2, -2, 0, 0, 0, 0, 2> : 0110 0000 0000 1001

<2, 2, 0, 0, 0, 0, -2> = <2, 0, 0, -2> * <0, 0, 0, 0> : 0110 0000

<2, -2, 0, 0, 0, 0, 2> = <0, 0, 0, 0> * <2, 0, 0, 2> : 0000 1001

<4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 0> = <2, -2, 0, 0, 0, 0, 2> * <2, 2, 0, 0, 0, 0, -2> : 0000 1001 0110 0000

<2, -2, 0, 0, 0, 0, 2> = <0, 0, 0, 0> * <2, 0, 0, 2> : 0000 1001

<2, 2, 0, 0, 0, 0, -2> = <2, 0, 0, -2> * <0, 0, 0, 0> : 0110 0000

<4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 0> = <2, -2, 0, 0, 0, 0, -2> * <2, 2, 0, 0, 0, 0, 2> : 0000 0110 1001 0000

<2, -2, 0, 0, 0, 0, -2> = <0, 0, 0, 0> * <2, 0, 0, -2> : 0000 0110

<2, 2, 0, 0, 0, 0, 2> = <2, 0, 0, 2> * <0, 0, 0, 0> : 1001 0000

```
===>   (1001 0000 0000 0110 0000 1001 0110 0000)
       (1001 0000 0000 0110 0000 0110 1001 0000)
       (0110 0000 0000 1001 0000 1001 0110 0000)
       (0110 0000 0000 1001 0000 0110 1001 0000)


<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 4> * <4, 0,
0, 0, 0, -4, 0, 0, 0, 0, -4>
        <4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 4> = <2, 2, 0, 0, 0, 0, 2> * <2, -2, 0, 0, 0, 0, 2> :
1001 0000 0000 1001
              <2, 2, 0, 0, 0, 0, 2> = <2, 0, 0, 2> * <0, 0, 0, 0> : 1001 0000
              <2, -2, 0, 0, 0, 0, 2> =<0, 0, 0, 0> * <2, 0, 0, 2> : 0000 1001

       <4, 0, 0, 0, 0, -4, 0, 0, 0, 0, -4> = <2, -2, 0, 0, 0, 0, -2> * <2, 2, 0, 0, 0, 0, -2>
: 0000 0110 0110 0000
              <2, -2, 0, 0, 0, 0, -2> = <0, 0, 0, 0> * <2, 0, 0, -2> : 0000 0110
              <2, 2, 0, 0, 0, 0, -2> = <2, 0, 0, -2> * <0, 0, 0, 0> : 0110 0000

===> (1001 0000 0000 1001 0000 0110 0110 0000)


<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0> * <4, 0,
0, 0, 0, 0, 0, 0, -4, 0, 0, 0>
       <4, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0> = <2, 0, 0, 2, -2, 0, 0> * <2, 0, 0, -2, 2, 0, 0> :
0010 1000 0100 0001
              <2, 0, 0, 2, -2, 0, 0> = <1, -1, 1, -1> * <1, 1, 1, 1> : 0010 1000
              <2, 0, 0, -2, 2, 0, 0> = <1, 1, -1, -1> * <1, -1, -1, 1> : 0100 0001

       <4, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0> = <2, 0, 0, 2, 2, 0, 0> * <2, 0, 0, -2, -2, 0, 0> :
1000 0010 0001 0100
              <2, 0, 0, 2, 2, 0, 0> = <1, 1, 1, 1> * <1, -1, 1, -1> : 1000 0010
              <2, 0, 0, -2, -2, 0, 0> = <1, -1, -1, 1> * <1, 1, -1, -1> : 0001 0100

       <4, 0, 0, 0, 0, 0, 0, -4, 0, 0, 0> = <2, 0, 0, -2, -2, 0, 0> * <2, 0, 0, 2, 2, 0, 0> :
0001 0100 1000 0010
              <2, 0, 0, -2, -2, 0, 0> = <1, -1, -1, 1> * <1, 1, -1, -1> = 0001 0100
              <2, 0, 0, 2, 2, 0, 0> = <1, 1, 1, 1> * <1, -1, 1, -1> = 1000 0010

       <4, 0, 0, 0, 0, 0, 0, -4, 0, 0, 0> = <2, 0, 0, -2, 2, 0, 0> * <2, 0, 0, 2, -2, 0, 0> :
0100 0001 0010 1000
              <2, 0, 0, -2, 2, 0, 0> = <1, 1, -1, -1> * <1, -1, -1, 1> : 0100 0001
              <2, 0, 0, 2, -2, 0, 0> = <1, -1, 1, -1> * <1, 1, 1, 1> : 0010 1000

===>   (0010 1000 0100 0001 0001 0100 1000 0010)
       (0010 1000 0100 0001 0100 0001 0010 1000)
       (1000 0010 0001 0100 0001 0100 1000 0010)
       (1000 0010 0001 0100 0100 0001 0010 1000)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, -4, -4, 0, 0> * <4, 0,
0, 0, 0, 0, 0, 0, 4, 4, 0, 0>
       <4, 0, 0, 0, 0, 0, 0, -4, -4, 0, 0> = <2, 0, 0, -2, -2, 0, 0> * <2, 0, 0, 2, -2, 0, 0>
: 0001 0100 0010 1000
              <2, 0, 0, -2, -2, 0, 0> = <1, -1, -1, 1> * <1, 1, -1, -1> : 0001 0100
              <2, 0, 0, 2, -2, 0, 0> = <1, -1, 1, -1> * <1, 1, 1, 1> : 0010 1000



       <4, 0, 0, 0, 0, 0, 0, 4, 4, 0, 0> = <2, 0, 0, 2, 2, 0, 0> * <2, 0, 0, -2, 2, 0, 0> :
1000 0010 0100 0001
```

<2, 0, 0, 2, 2, 0, 0> = <1, 1, 1, 1> * <1, -1, 1, -1> : 1000 0010

<2, 0, 0, -2, 2, 0, 0> = <1, 1, -1, -1> * <1, -1, -1, 1> : 0100 0001

===> (0001 0100 0010 1000 1000 0010 0100 0001)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 0, -4, 0, 0> * <4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0>

<4, 0, 0, 0, 0, 0, 0, 0, -4, 0, 0> = <2, 0, 0, 0, -2, -2, 2> * <2, 0, 0, 0, -2, 2, -2> : 0001 1000 0010 0100

<2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

<2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<4, 0, 0, 0, 0, 0, 0, 0, -4, 0, 0> = <2, 0, 0, 0, -2, 2, -2> * <2, 0, 0, 0, -2, -2, 2> : 0010 0100 0001 1000

<2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

<4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0> = <2, 0, 0, 0, 2, -2, -2> * <2, 0, 0, 0, 2, 2, 2> : 0100 0010 1000 0001

<2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

<4, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0> = <2, 0, 0, 0, 2, 2, 2> * <2, 0, 0, 0, 2, -2, -2> : 1000 0001 0100 0010

<2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

<2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

===> (0001 1000 0010 0100 0100 0010 1000 0001)

(0001 1000 0010 0100 1000 0001 0100 0010)

(0010 0100 0001 1000 0100 0010 1000 0001)

(0010 0100 0001 1000 1000 0001 0100 0010)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0> * <4, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0>

<4, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0> = <2, 0, 0, 0, 2, 2, 2> * <2, 0, 0, 0, -2, 2, -2> : 1000 0001 0010 0100

<2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

<2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<4, 0, 0, 0, 0, 0, 0, 0, 4, 0> = <2, 0, 0, 0, -2, 2, -2> * <2, 0, 0, 0, 2, 2, 2> : 0010 0100 1000 0001

<2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

<4, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0> = <2, 0, 0, 0, -2, -2, 2> * <2, 0, 0, 0, 2, -2, -2> : 0001 1000 0100 0010

<2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

<2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<4, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0> = <2, 0, 0, 0, 2, -2, -2> * <2, 0, 0, 0, -2, -2, 2> : 0100 0010 0001 1000

<2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

```
===>    (1000 0001 0010 0100 0001 1000 0100 0010)
        (1000 0001 0010 0100 0100 0010 0001 1000)
        (0010 0100 1000 0001 0001 1000 0100 0010)
        (0010 0100 1000 0001 0100 0010 0001 1000)


<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4><2) * <4,
0, 0, 0, 0, 0, 0, 0, 0, 0, -4><2)
       <4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4> = <2, 0, 0, 0, 2, 2, 2> * <2, 0, 0, 0, -2, -2, 2> :
1000 0001 0001 1000
               <2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001
               <2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000


       <4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4> = <2, 0, 0, 0, -2, -2, 2> * <2, 0, 0, 0, 2, 2, 2> :
0001 1000 1000 0001
               <2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000
               <2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001


       <4, 0, 0, 0, 0, 0, 0, 0, 0, 0, -4> = <2, 0, 0, 0, 2, -2, -2> * <2, 0, 0, 0, -2, 2, -2>
: 0100 0010 0010 0100
               <2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010
               <2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100


       <4, 0, 0, 0, 0, 0, 0, 0, 0, 0, -4> = <2, 0, 0, 0, -2, 2, -2> * <2, 0, 0, 0, 2, -2, -2>
: 0010 0100 0100 0010
               <2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100
               <2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010


===> (1000 0001 0001 1000 0100 0010 0010 0100)
        (1000 0001 0001 1000 0010 0100 0100 0010)
        (0001 1000 1000 0001 0100 0010 0010 0100)
        (0001 1000 1000 0001 0010 0100 0100 0010)


<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 4, -4, 0, 0> * <4, 0,
0, 0, 0, 0, 0, -4, 4, 0, 0>
        <4, 0, 0, 0, 0, 0, 0, 4, -4, 0, 0> = <2, 0, 0, 2, -2, 0, 0> * <2, 0, 0, -2, -2, 0, 0>
: 0010 1000 0001 0100
               <2, 0, 0, 2, -2, 0, 0> = <1, -1, 1, -1> * <1, 1, 1, 1> : 0010 1000
               <2, 0, 0, -2, -2, 0, 0> = <1, -1, -1, 1> * <1, 1, -1, -1> : 0001 0100


        <4, 0, 0, 0, 0, 0, 0, -4, 4, 0, 0> = <2, 0, 0, -2, 2, 0, 0> * <2, 0, 0, 2, 2, 0, 0> :
0100 0001 1000 0010
               <2, 0, 0, -2, 2, 0, 0> = <1, 1, -1, -1> * <1, -1, -1, 1> : 0100 0001
               <2, 0, 0, 2, 2, 0, 0> = <1, 1, 1, 1> * <1, -1, 1, -1> : 1000 0010


===> (0010 1000 0001 0100 0100 0001 1000 0010)


<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, -4, 0, 0, 0> * <4, 0,
0, 0, 0, 0, 0, 4, 0, 0, 0>
       <4, 0, 0, 0, 0, 0, 0, -4, 0, 0, 0> = <2, 0, 0, -2, 2, 0, 0> * <2, 0, 0, 2, -2, 0, 0> :
0100 0001 0010 1000
               <2, 0, 0, -2, 2, 0, 0> = <1, 1, -1, -1> * <1, -1, -1, 1> : 0100 0001
               <2, 0, 0, 2, -2, 0, 0> = <1, -1, 1, -1> * <1, 1, 1, 1> : 0010 1000


       <4, 0, 0, 0, 0, 0, 0, -4, 0, 0, 0> = <2, 0, 0, -2, -2, 0, 0> * <2, 0, 0, 2, 2, 0, 0> :
0001 0100 1000 0010
               <2, 0, 0, -2, -2, 0, 0> = <1, -1, -1, 1> * <1, 1, -1, -1> : 0001 0100
               <2, 0, 0, 2, 2, 0, 0> = <1, 1, 1, 1> * <1, -1, 1, -1> : 1000 0010
```

```
        ---
        <4, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0> = <2, 0, 0, 2, -2, 0, 0> * <2, 0, 0, -2, 2, 0, 0> :
0010 1000 0010 1000
                <2, 0, 0, 2, -2, 0, 0> = <1, -1, 1, -1> * <1, 1, 1, 1> : 0010 1000
                <2, 0, 0, -2, 2, 0, 0> = <1, 1, -1, -1> * <1, -1, -1, 1> : 0100 0001

        <4, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0> = <2, 0, 0, 2, 2, 0, 0> * <2, 0, 0, -2, -2, 0, 0> :
1000 0010 0001 0100
                <2, 0, 0, 2, 2, 0, 0> = <1, 1, 1, 1> * <1, -1, 1, -1> : 1000 0010
                <2, 0, 0, -2, -2, 0, 0> = <1, -1, -1, 1> * <1, 1, -1, -1> : 0001 0100

===>    (0100 0001 0010 1000 0010 1000 0010 1000)
        (0100 0001 0010 1000 1000 0010 0001 0100)
        (0001 0100 1000 0010 0010 1000 0010 1000)
        (0001 0100 1000 0010 1000 0010 0001 0100)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, -4, 0, 0, -4, 0> * <4, 0,
0, 0, 0, 0, 4, 0, 0, 4, 0>
        <4, 0, 0, 0, 0, 0, -4, 0, 0, -4, 0> = <2, 0, -2, 0, 0, -2, 0> * <2, 0, 2, 0, 0, -2, 0>
: 0001 0010 0100 1000
                <2, 0, -2, 0, 0, -2, 0> = <1, -1, -1, 1> * <1, -1, 1, -1> : 0001 0010
                <2, 0, 2, 0, 0, -2, 0> = <1, 1, -1, -1> * <1, 1, 1, 1> : 0100 1000

        <4, 0, 0, 0, 0, 0, 4, 0, 0, 4, 0> = <2, 0, 2, 0, 0, 2, 0> * <2, 0, -2, 0, 0, 2, 0> :
1000 0100 0010 0001
                <2, 0, 2, 0, 0, 2, 0> = <1, 1, 1, 1> * <1, 1, -1, -1> : 1000 0100
                <2, 0, -2, 0, 0, 2, 0> = <1, -1, 1, -1> * <1, -1, -1, 1> : 0010 0001

==> (0001 0010 0100 1000 1000 0100 0010 0001)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 0> * <4, 0,
0, 0, 0, 4, 0, 0, 0, 0, 0>
        <4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 0> = <2, -2, 0, 0, 0, 0, 2> * <2, 2, 0, 0, 0, 0, -2> :
0000 1001 0110 0000
                <2, -2, 0, 0, 0, 0, 2> = <0, 0, 0, 0> * <2, 0, 0, 2> : 0000 1001
                <2, 2, 0, 0, 0, 0, -2> = <2, 0, 0, -2> * <0, 0, 0, 0> : 0110 0000

        <4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 0> = <2, -2, 0, 0, 0, 0, -2> * <2, 2, 0, 0, 0, 0, 2> :
0000 0110 1001 0000
                <2, -2, 0, 0, 0, 0, -2> = <0, 0, 0, 0> * <2, 0, 0, -2> : 0000 0110
                <2, 2, 0, 0, 0, 0, 2> = <2, 0, 0, 2> * <0, 0, 0, 0> : 1001 0000

        <4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0> = <2, 2, 0, 0, 0, 0, 2> * <2, -2, 0, 0, 0, 0, -2> :
1001 0000 0000 0110
                <2, 2, 0, 0, 0, 0, 2> = <2, 0, 0, 2> * <0, 0, 0, 0> : 1001 0000
                <2, -2, 0, 0, 0, 0, -2> = <0, 0, 0, 0> * <2, 0, 0, -2> : 0000 0110

        <4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0> = <2, 2, 0, 0, 0, 0, -2> * <2, -2, 0, 0, 0, 0, 2> :
0110 0000 0000 1001
                <2, 2, 0, 0, 0, 0, -2> = <2, 0, 0, -2> * <0, 0, 0, 0> : 0110 0000
                <2, -2, 0, 0, 0, 0, 2> = <0, 0, 0, 0> * <2, 0, 0, 2> : 0000 1001

===>    (0000 1001 0110 0000 1001 0000 0000 0110)
        (0000 1001 0110 0000 0110 0000 0000 1001)
        (0000 0110 1001 0000 1001 0000 0000 0110)
```

(0000 0110 1001 0000 0110 0000 0000 1001)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 4, 4, 0, 0> * <4, 0, 0, 0, 0, 0, 0, -4, -4, 0, 0>

<4, 0, 0, 0, 0, 0, 0, 4, 4, 0, 0> = <2, 0, 0, 2, 2, 0, 0> * <2, 0, 0, -2, 2, 0, 0> : 1000 0010 0100 0001

<2, 0, 0, 2, 2, 0, 0> = <1, 1, 1, 1> * <1, -1, 1, -1> : 1000 0010

<2, 0, 0, -2, 2, 0, 0> = <1, 1, -1, -1> * <1, -1, -1, 1> : 0100 0001

<4, 0, 0, 0, 0, 0, 0, -4, -4, 0, 0> = <2, 0, 0, -2, -2, 0, 0> * <2, 0, 0, 2, -2, 0, 0> : 0001 0100 0010 1000

<2, 0, 0, -2, -2, 0, 0> = <1, -1, -1, 1> * <1, 1, -1, -1> : 0001 0100

<2, 0, 0, 2, -2, 0, 0> = <1, -1, 1, -1> * <1, 1, 1, 1> : 0010 1000

===> (1000 0010 0100 0001 0001 0100 0010 1000)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, -4, 0, 0, 0, 0, -4> * <4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 4>

<4, 0, 0, 0, 0, -4, 0, 0, 0, 0, -4> = <2, -2, 0, 0, 0, 0, -2> * <2, 2, 0, 0, 0, 0, -2> : 0000 0110 0110 0000

<2, -2, 0, 0, 0, 0, -2> = <0, 0, 0, 0> * <2, 0, 0, -2> : 0000 0110

<2, 2, 0, 0, 0, 0, -2> = <2, 0, 0, -2> * <0, 0, 0, 0> : 0110 0000

<4, 0, 0, 0, 0, 4, 0, 0, 0, 0, 4> = <2, 2, 0, 0, 0, 0, 2> * <2, -2, 0, 0, 0, 0, 2> : 1001 0000 0000 1001

<2, 2, 0, 0, 0, 0, 2> = <2, 0, 0, 2> * <0, 0, 0, 0> : 1001 0000

<2, -2, 0, 0, 0, 0, 2> =<0, 0, 0, 0> * <2, 0, 0, 2> : 0000 1001

===> (0000 0110 0110 0000 1001 0000 0000 1001)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 4, 0, 0> * <4, 0, 0, 0, 0, 0, 0, -4, 0, 0>

<4, 0, 0, 0, 0, 0, 0, 4, 0, 0> = <2, 0, 0, 2, -2, -2> * <2, 0, 0, 2, 2, 2> : 0100 0010 1000 0001

<2, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<2, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

<4, 0, 0, 0, 0, 0, 0, 4, 0, 0> = <2, 0, 0, 2, 2, 2> * <2, 0, 0, 2, -2, -2> : 1000 0001 0100 0010

<2, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

<2, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<4, 0, 0, 0, 0, 0, 0, -4, 0, 0> = <2, 0, 0, -2, -2, 2> * <2, 0, 0, -2, 2, -2> : 0001 1000 0010 0100

<2, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

<2, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<4, 0, 0, 0, 0, 0, 0, -4, 0, 0> = <2, 0, 0, -2, 2, -2> * <2, 0, 0, -2, -2, 2> : 0010 0100 0001 1000

<2, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<2, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

===> (0100 0010 1000 0001 0001 1000 0010 0100)

(0100 0010 1000 0001 0010 0100 0001 1000)

(1000 0001 0100 0010 0001 1000 0010 0100)

(1000 0001 0100 0010 0010 0100 0001 1000)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 0, 0, 0, -4> * <4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4>

<4, 0, 0, 0, 0, 0, 0, 0, 0, 0, -4> = <2, 0, 0, 0, 2, -2, -2> * <2, 0, 0, 0, -2, 2, -2> : 0100 0010 0010 0100

<2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<4, 0, 0, 0, 0, 0, 0, 0, 0, 0, -4> = <2, 0, 0, 0, -2, 2, -2> * <2, 0, 0, 0, 2, -2, -2> : 0010 0100 0100 0010

<2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4> = <2, 0, 0, 0, 2, 2, 2> * <2, 0, 0, 0, -2, -2, 2> : 1000 0001 0001 1000

<2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

<2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

<4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4> = <2, 0, 0, 0, -2, -2, 2> * <2, 0, 0, 0, 2, 2, 2> : 0001 1000 1000 0001

<2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

<2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

===> (0100 0010 0010 0100 1000 0001 0001 1000)

(0100 0010 0010 0100 0001 1000 1000 0001)

(0010 0100 0100 0010 1000 0001 0001 1000)

(0010 0100 0100 0010 0001 1000 1000 0001)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 4> * <4, 0, 0, 0, 0, 4, 0, 0, 0, 0, -4>

<4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 4> = <2, -2, 0, 0, 0, 0, 2> * <2, 2, 0, 0, 0, 0, 2> : 0000 1001 1001 0000

<2, -2, 0, 0, 0, 0, 2> = <0, 0, 0, 0> * <2, 0, 0, 2> : 0000 1001

<2, 2, 0, 0, 0, 0, 2> = <2, 0, 0, 2> * <0, 0, 0, 0> : 1001 0000

<4, 0, 0, 0, 0, 4, 0, 0, 0, 0, -4> = <2, 2, 0, 0, 0, 0, -2> * <2, -2, 0, 0, 0, 0, -2> : 0110 0000 0000 0110

<2, 2, 0, 0, 0, 0, -2> = <2, 0, 0, -2> * <0, 0, 0, 0> : 0110 0000

<2, -2, 0, 0, 0, 0, -2> = <0, 0, 0, 0> * <2, 0, 0, -2> : 0000 0110

===> (0000 1001 1001 0000 0110 0000 0000 0110)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0> * <4, 0, 0, 0, 0, 0, -4, 0, 0, 0, 0>

<4, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0> = <2, 0, 2, 0, 0, 2, 0> * <2, 0, -2, 0, 0, -2, 0> : 1000 0100 0001 0010

<2, 0, 2, 0, 0, 2, 0> = <1, 1, 1, 1> * <1, 1, -1, -1> : 1000 0100

<2, 0, -2, 0, 0, -2, 0> = <1, -1, -1, 1> * <1, -1, 1, -1> : 0001 0010

<4, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0> = <2, 0, 2, 0, 0, -2, 0> * <2, 0, -2, 0, 0, 2, 0> : 0100 1000 0010 0001

<2, 0, 2, 0, 0, -2, 0> = <1, 1, -1, -1> * <1, 1, 1, 1> : 0100 1000

<2, 0, -2, 0, 0, 2, 0> = <1, -1, 1, -1> * <1, -1, -1, 1> : 0010 0001

<4, 0, 0, 0, 0, 0, -4, 0, 0, 0, 0> = <2, 0, -2, 0, 0, -2, 0> * <2, 0, 2, 0, 0, 2, 0> : 0001 0010 1000 0100

<2, 0, -2, 0, 0, -2, 0> = <1, -1, -1, 1> * <1, -1, 1, -1> : 0001 0010

<2, 0, 2, 0, 0, 2, 0> = <1, 1, 1, 1> * <1, 1, -1, -1> : 1000 0100

<4, 0, 0, 0, 0, 0, -4, 0, 0, 0, 0> = <2, 0, -2, 0, 0, 2, 0> * <2, 0, 2, 0, 0, -2, 0> : 0010 0001 0100 1000

<2, 0, -2, 0, 0, 2, 0> = <1, -1, 1, -1> * <1, -1, -1, 1> : 0010 0001

<2, 0, 2, 0, 0, -2, 0> = <1, 1, -1, -1> * <1, 1, 1, 1> : 0100 1000

===> (1000 0100 0001 0010 0001 0010 1000 0100)

(1000 0100 0001 0010 0010 0001 0100 1000)

(0100 1000 0010 0001 0001 0010 1000 0100)

(0100 1000 0010 0001 0010 0001 0100 1000)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0> * <4, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0>

<4, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0> = <2, 0, 0, 0, -2, -2, 2> * <2, 0, 0, 0, 2, -2, -2> : 0001 1000 0100 0010

<2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

<2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<4, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0> = <2, 0, 0, 0, 2, -2, -2> * <2, 0, 0, 0, -2, -2, 2> : 0100 0010 0001 1000

<2, 0, 0, 0, 2, -2, -2> = <1, 1, -1, -1> * <1, -1, 1, -1> : 0100 0010

<2, 0, 0, 0, -2, -2, 2> = <1, -1, -1, 1> * <1, 1, 1, 1> : 0001 1000

<4, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0> = <2, 0, 0, 0, 2, 2, 2> * <2, 0, 0, 0, -2, 2, -2> : 1000 0001 0010 0100

<2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

<2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<4, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0> = <2, 0, 0, 0, -2, 2, -2> * <2, 0, 0, 0, 2, 2, 2> : 0010 0100 1000 0001

<2, 0, 0, 0, -2, 2, -2> = <1, -1, 1, -1> * <1, 1, -1, -1> : 0010 0100

<2, 0, 0, 0, 2, 2, 2> = <1, 1, 1, 1> * <1, -1, -1, 1> : 1000 0001

===> (0001 1000 0100 0010 1000 0001 0010 0100)

(0001 1000 0100 0010 0010 0100 1000 0001)

(0100 0010 0001 1000 1000 0001 0010 0100)

(0100 0010 0001 1000 0010 0100 1000 0001)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 0, -4, 4, 0, 0> * <4, 0, 0, 0, 0, 0, 0, 4, -4, 0, 0>

<4, 0, 0, 0, 0, 0, 0, -4, 4, 0, 0> = <2, 0, 0, -2, 2, 0, 0> * <2, 0, 0, 2, 2, 0, 0> : 0100 0001 1000 0010

<2, 0, 0, -2, 2, 0, 0> = <1, 1, -1, -1> * <1, -1, -1, 1> : 0100 0001

<2, 0, 0, 2, 2, 0, 0> = <1, 1, 1, 1> * <1, -1, 1, -1> : 1000 0010

<4, 0, 0, 0, 0, 0, 0, 4, -4, 0, 0> = <2, 0, 0, 2, -2, 0, 0> * <2, 0, 0, -2, -2, 0, 0> : 0010 1000 0001 0100

<2, 0, 0, 2, -2, 0, 0> = <1, -1, 1, -1> * <1, 1, 1, 1> : 0010 1000

<2, 0, 0, -2, -2, 0, 0> = <1, -1, -1, 1> * <1, 1, -1, -1> : 0001 0100

===> (0100 0001 1000 0010 0010 1000 0001 0100)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 4, 0, 0, 0, 0, -4> * <4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 4>

<4, 0, 0, 0, 0, 4, 0, 0, 0, 0, -4> = <2, 2, 0, 0, 0, 0, -2> * <2, -2, 0, 0, 0, 0, -2> : 0110 0000 0000 0110

<2, 2, 0, 0, 0, 0, -2> = <2, 0, 0, -2> * <0, 0, 0, 0> : 0110 0000

<2, -2, 0, 0, 0, 0, -2> = <0, 0, 0, 0> * <2, 0, 0, -2> : 0000 0110

<4, 0, 0, 0, 0, -4, 0, 0, 0, 0, 4> = <2, -2, 0, 0, 0, 0, 2> * <2, 2, 0, 0, 0, 0, 2> : 0000 1001 1001 0000

<2, -2, 0, 0, 0, 0, 2> = <0, 0, 0, 0> * <2, 0, 0, 2> : 0000 1001

<2, 2, 0, 0, 0, 0, 2> = <2, 0, 0, 2> * <0, 0, 0, 0> : 1001 0000

===> (0110 0000 0000 0110 0000 1001 1001 0000)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, 4, 0, 0, 4, 0> * <4, 0, 0, 0, 0, 0, -4, 0, 0, -4, 0>

<4, 0, 0, 0, 0, 0, 4, 0, 0, 4, 0> = <2, 0, 2, 0, 0, 2, 0> * <2, 0, -2, 0, 0, 2, 0> : 1000 0100 0010 0001

<2, 0, 2, 0, 0, 2, 0> = <1, 1, 1, 1> * <1, 1, -1, -1> : 1000 0100

<2, 0, -2, 0, 0, 2, 0> = <1, -1, 1, -1> * <1, -1, -1, 1> : 0010 0001

<4, 0, 0, 0, 0, 0, -4, 0, 0, -4, 0> = <2, 0, -2, 0, 0, -2, 0> * <2, 0, 2, 0, 0, -2, 0> : 0001 0010 0100 1000

<2, 0, -2, 0, 0, -2, 0> = <1, -1, -1, 1> * <1, -1, 1, -1> : 0001 0010

<2, 0, 2, 0, 0, -2, 0> = <1, 1, -1, -1> * <1, 1, 1, 1> : 0100 1000

===> (1000 0100 0010 0001 0001 0010 0100 1000)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, -4, 0, 0, 0, 0> * <4, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0>

<4, 0, 0, 0, 0, 0, -4, 0, 0, 0, 0> = <2, 0, -2, 0, 0, -2, 0> * <2, 0, 2, 0, 0, 2, 0> : 0001 0010 1000 0100

<2, 0, -2, 0, 0, -2, 0> = <1, -1, -1, 1> * <1, -1, 1, -1> : 0001 0010

<2, 0, 2, 0, 0, 2, 0> = <1, 1, 1, 1> * <1, 1, -1, -1> : 1000 0100

<4, 0, 0, 0, 0, 0, -4, 0, 0, 0, 0> = <2, 0, -2, 0, 0, 2, 0> * <2, 0, 2, 0, 0, -2, 0> : 0010 0001 0100 1000

<2, 0, -2, 0, 0, 2, 0> = <1, -1, 1, -1> * <1, -1, -1, 1> : 0010 0001

<2, 0, 2, 0, 0, -2, 0> = <1, 1, -1, -1> * <1, 1, 1, 1> : 0100 1000

<4, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0> = <2, 0, 2, 0, 0, 2, 0> * <2, 0, -2, 0, 0, -2, 0> : 1000 0100 0001 0010

<2, 0, 2, 0, 0, 2, 0> = <1, 1, 1, 1> * <1, 1, -1, -1> : 1000 0100

<2, 0, -2, 0, 0, -2, 0> = <1, -1, -1, 1> * <1, -1, 1, -1> : 0001 0010

<4, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0> = <2, 0, 2, 0, 0, -2, 0> * <2, 0, -2, 0, 0, 2, 0> : 0100 1000 0010 0001

<2, 0, 2, 0, 0, -2, 0> = <1, 1, -1, -1> * <1, 1, 1, 1> : 0100 1000

<2, 0, -2, 0, 0, 2, 0> = <1, -1, 1, -1> * <1, -1, -1, 1> : 0010 0001

===> (0001 0010 1000 0100 1000 0100 0001 0010)

(0001 0010 1000 0100 0100 1000 0010 0001)

(0010 0001 0100 1000 1000 0100 0001 0010)

(0010 0001 0100 1000 0100 1000 0010 0001)

<8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0> = <4, 0, 0, 0, 0, 0, -4, 0, 0, 4, 0> * <4, 0, 0, 0, 0, 0, 4, 0, 0, -4, 0>

```
       <4, 0, 0, 0, 0, 0, -4, 0, 0, 4, 0> = <2, 0, -2, 0, 0, 2, 0> * <2, 0, 2, 0, 0, 2, 0> :
0010 0001 1000 0100
               <2, 0, -2, 0, 0, 2, 0> = <1, -1, 1, -1> * <1, -1, -1, 1> : 0010 0001
               <2, 0, 2, 0, 0, 2, 0>  = <1, 1, 1, 1> * <1, 1, -1, -1> : 1000 0100

       <4, 0, 0, 0, 0, 0, 4, 0, 0, -4, 0> = <2, 0, 2, 0, 0, -2, 0> * <2, 0, -2, 0, 0, -2, 0> :
0100 1000 0001 0010
               <2, 0, 2, 0, 0, -2, 0>  = <1, 1, -1, -1> * <1, 1, 1, 1> : 0100 1000
               <2, 0, -2, 0, 0, -2, 0> = <1, -1, -1, 1> * <1, -1, 1, -1> : 0001 0010

===> (0010 0001 1000 0100 0100 1000 0001 0010)
```

ANEXO III - FUNCIONES BOOLEANAS OBTENIDAS POR LA APLICACIÓN

El objetivo de este anexo es exponer los resultados obtenidos por la aplicación, pero expresados como funciones Booleanas. Dado que el tamaño de las funciones crece a medida que aumenta el orden de correlación reducimos la exposición a N de 2 a 5.

N2 K1

(0 1 1 0)

(1 0 0 1)

N2 K2

(1 1 1 1)

N3 K1

(1 0 0 0 0 0 0 1)

(0 0 0 1 1 0 0 0)

(0 0 1 0 0 1 0 0)

(0 1 0 0 0 0 1 0)

N3 K2

(1 0 0 1 0 1 1 0)

(0 1 1 0 1 0 0 1)

N3 K3

(1 1 1 1 1 1 1 1)

N4 K1

(0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0)

(0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0)

(0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0)

(0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0)

(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)

(0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0)

(0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0)

(0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0)

N4 K2

(1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0)

(0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1)

(1 0 0 1 1 0 0 1 0 1 1 0 0 1 1 0)

(0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1)

(0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 1)

(1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 0)

(1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0)

(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)

(0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0)

(0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1)

N4 K3

(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
(0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0)

N4 K4

(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)

N5 K1

(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 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0)
(0 1 0 0 0 0 0 0 0 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 0 0 1 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)
(0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0)
(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)
(0 0 1 0 0 0 0 0 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 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
(0 0 0 1 0 0 0 0 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 0 0 1 0 0 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 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 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 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0)

N5 K2

(1 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0)
(0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0)
(0 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0)
(0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0)
(0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0)
(1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0)
(0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1)
(0 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 1)
(1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0)
(0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0)
(1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0)
(1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0)
(0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0)
(0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0)
(0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1)
(1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0)
(0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0)
(0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0)
(1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 0)
(0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0)
(0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1)

(1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0)
(0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1)
(1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0)
(0 1 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0)
(1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0)
(0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 1)
(0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0)
(0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0)
(0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1)
(0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1)
(0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0)
(0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1)
(0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0)
(0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0)
(0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 1 0)
(1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0)
(0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0)
(0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0)
(0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0)
(0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 0)
(0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1)
(0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0)
(0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 1)
(0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1)
(1 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 1 0 0 1 0 0 0 0)
(0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 1)
(0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 0)
(1 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0)
(1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0)
(1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0)
(0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0)
(0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 1 0 0 1 0 0 0 0)
(0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1)
(0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0)
(0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1)
(0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0)
(0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 1 1 0 0 0)
(0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0)
(0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0)

N5 K3

(0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 0 1 1 0 0 1 1 0)
(1 0 0 1 1 0 0 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1)
(1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 1)
(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
(0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0)
(0 0 1 1 1 1 0 0 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 0 0 1 1 1 1 0 0)
(0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0)
(0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
(1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 0 0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1)
(0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 0)
(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0)

(1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1)

N5 K4

(0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1)
(1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0)

N5 K5

(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)

ANEXO IV - EXTENSIÓN DE CLASES DE *K* A *K+1*

Al extender las clases de k a k+1, hay que calcular todos los valores de Fourier de peso de Hamming k+1. Esto implica que si una clase de correlación de orden k y n variables tenía N funciones, ahora estas N funciones se van a dividir en diferentes clases dependiendo del valor de las transformadas de Fourier de peso de Hamming k+1.

Por tal motivo nos interesa saber no sólo cual es el valor de estas transformadas para entender sus simetrías, sino que también entender cuántas funciones de las N originales caen en cada clase.

De esta manera, se quiere tratar de entender las simetrías y ver la manera de poder generalizar el concepto de clases normales (definidas para el caso k=1) para k general, y poder así usar menos memoria y avanzar más en la tabla a fin de encontrar más datos.

Caso A

Se probó extender un grado más las clases dentro de los rangos N [1…13] y K [1…13] y que cumplen con la siguiente particularidad para cualquier $p \in Naturales$:

$$w(n,k) = 2^{n-k-1} \Rightarrow w(n+p,k+p) = 2^{n-k-1} \quad con\ w(n,k) = \frac{W(n,k)}{2^k}$$

En los resultados obtenidos se notó que, en la extensión, algunas clases resultado representaban a más de una función, esto se vio reflejado en el cálculo con disminución en tiempos de cálculo de clases a partir de las predecesoras [N-1, K-1].

A continuación, mostraremos extensiones de ejemplo de este caso:

N6 K4 W32 ext 4

Podemos ver que estamos en el caso A ya que se cumple la siguiente igualdad:

$$\frac{32}{2^4} = 2 = 2^{6-4-1}$$

Cantidad funciones	Clase	
14	<32	... {56 0's} ...>

Tenemos una clase de equivalencia que contiene 14 funciones.

N6 K4 W32 ext 5

Al extender la clase de equivalencia a k+1 (5) se obtienen las siguientes clases de equivalencia:

Cantidad funciones	Clases								
2	<32	... {56 0's} ...	0	0	0	0	0	0	0>
1	<32	... {56 0's} ...	0	0	0	0	0	0	32>
1	<32	... {56 0's} ...	0	0	0	0	0	0	-32>
1	<32	... {56 0's} ...	0	0	0	0	0	32	0>
1	<32	... {56 0's} ...	0	0	0	0	0	-32	0>
1	<32	... {56 0's} ...	0	0	0	0	32	0	0>
1	<32	... {56 0's} ...	0	0	0	0	-32	0	0>
1	<32	... {56 0's} ...	0	0	0	32	0	0	0>
1	<32	... {56 0's} ...	0	0	0	-32	0	0	0>
1	<32	... {56 0's} ...	0	0	32	0	0	0	0>
1	<32	... {56 0's} ...	0	0	-32	0	0	0	0>
1	<32	... {56 0's} ...	0	32	0	0	0	0	0>
1	<32	... {56 0's} ...	-32	0	0	0	0	0	0>

En total tenemos 13 clases de equivalencia que se agrupan de la siguiente manera: tenemos 1 clase de equivalencia que contiene 2 funciones Booleanas y tenemos 12 clases de equivalencia que cada una contiene 1 función Booleana.

N3 K1 W4 ext 1

Podemos ver que estamos en el caso A ya que se cumple la siguiente igualdad:

$$\frac{4}{2^1} = 2 = 2^{3-1-1}$$

Cantidad funciones	Clase			
8	<4	0	0	0>

Tenemos 1 clase de equivalencia que contiene 8 funciones Booleanas.

N3 K1 W4 ext 2

Al extender la clase de equivalencia a k+1 (2) se obtienen las siguientes clases de equivalencia:

Cantidad funciones	Clases						
2	<4	0	0	0	0	0	0>
1	<4	0	0	0	-4	0	0>
1	<4	0	0	0	4	0	0>
1	<4	0	0	0	0	4	0>
1	<4	0	0	0	0	0	4>
1	<4	0	0	0	0	-4	0>
1	<4	0	0	0	0	0	-4>

En total tenemos 7 clases de equivalencia que se agrupan de la siguiente manera: 6 clases de equivalencia que cada una contiene solamente una función Booleana y además tenemos 1 clase de equivalencia que contiene 2 funciones Booleanas.

N3 K2 W4 ext 2

Podemos ver que estamos en el caso A ya que se cumple la siguiente igualdad:

$$\frac{4}{2^2} = 1 = 2^{3-2-1}$$

Cantidad funciones	Clase						
2	<4	0	0	0	0	0	0>

Tenemos una clase de equivalencia que contiene 2 funciones Booleanas.

N3 K2 W4 ext 3

Al extender la clase de equivalencia a k+1 (3) se obtienen las siguientes clases de equivalencia:

Cantidad funciones	Clase							
1	<4	0	0	0	0	0	0	-4>
1	<4	0	0	0	0	0	0	4>

Tenemos 2 clases de equivalencia que cada una contiene 2 funciones Booleanas.

N4 K1 W8 ext 1

Podemos ver que estamos en el caso A ya que se cumple la siguiente igualdad:

$$\frac{8}{2^1} = 4 = 2^{4-1-1}$$

Cantidad funciones	Clase				
222	<8	0	0	0	0>

Tenemos 1 clase de equivalencia que contiene 222 funciones Booleanas.

N4 K1 W8 ext 2

Al extender la clase de equivalencia a k+1 (2) se obtienen las siguientes clases de equivalencia:

Cantidad funciones	Clases										
10	<8	0	0	0	0	0	0	0	0	0	0>
4	<8	0	0	0	0	0	0	0	0	-4	0>
4	<8	0	0	0	0	0	-4	0	0	0	0>
4	<8	0	0	0	0	0	0	0	-4	0	0>
4	<8	0	0	0	0	0	4	0	0	0	0>
4	<8	0	0	0	0	-4	0	0	0	0	0>
4	<8	0	0	0	0	0	0	0	0	0	4>
4	<8	0	0	0	0	0	0	4	0	0	0>
4	<8	0	0	0	0	0	0	0	0	4	0>
4	<8	0	0	0	0	0	0	-4	0	0	0>
4	<8	0	0	0	0	4	0	0	0	0	0>
4	<8	0	0	0	0	0	0	0	4	0	0>
4	<8	0	0	0	0	0	0	0	0	0	-4>
2	<8	0	0	0	0	0	0	0	4	4	0>
2	<8	0	0	0	0	4	0	0	0	-4	0>
2	<8	0	0	0	0	0	-4	4	0	0	0>
2	<8	0	0	0	0	4	0	0	4	0	0>
2	<8	0	0	0	0	4	-4	0	0	0	0>
2	<8	0	0	0	0	0	0	0	4	-4	0>
2	<8	0	0	0	0	0	0	0	0	-4	-4>
2	<8	0	0	0	0	-4	0	0	0	-4	0>
2	<8	0	0	0	0	0	4	0	0	0	4>
2	<8	0	0	0	0	0	0	0	4	0	-4>
2	<8	0	0	0	0	0	4	0	4	0	0>
2	<8	0	0	0	0	0	-4	-4	0	0	0>
2	<8	0	0	0	0	0	-4	0	-4	0	0>
2	<8	0	0	0	0	0	0	0	0	4	4>
2	<8	0	0	0	0	-4	-4	0	0	0	0>
2	<8	0	0	0	0	-4	0	0	0	4	0>
2	<8	0	0	0	0	0	-4	0	0	0	4>
2	<8	0	0	0	0	0	0	-4	0	4	0>
2	<8	0	0	0	0	0	0	0	-4	4	0>
2	<8	0	0	0	0	0	0	4	0	-4	0>
2	<8	0	0	0	0	-4	0	4	0	0	0>
2	<8	0	0	0	0	0	0	0	-4	-4	0>
2	<8	0	0	0	0	0	0	-4	0	0	4>

2	<8	0	0	0	0	4	0	0	0	4	0>
2	<8	0	0	0	0	0	0	4	0	0	4>
2	<8	0	0	0	0	0	4	0	0	0	-4>
2	<8	0	0	0	0	0	0	4	0	0	-4>
2	<8	0	0	0	0	4	0	-4	0	0	0>
2	<8	0	0	0	0	-4	0	0	-4	0	0>
2	<8	0	0	0	0	0	0	0	0	4	-4>
2	<8	0	0	0	0	0	0	0	-4	0	4>
2	<8	0	0	0	0	0	4	4	0	0	0>
2	<8	0	0	0	0	-4	0	0	4	0	0>
2	<8	0	0	0	0	0	-4	0	0	0	-4>
2	<8	0	0	0	0	-4	0	-4	0	0	0>
2	<8	0	0	0	0	4	0	0	-4	0	0>
2	<8	0	0	0	0	4	4	0	0	0	0>
2	<8	0	0	0	0	0	0	0	0	-4	4>
2	<8	0	0	0	0	0	0	0	4	0	4>
2	<8	0	0	0	0	-4	4	0	0	0	0>
2	<8	0	0	0	0	0	4	0	-4	0	0>
2	<8	0	0	0	0	0	0	4	0	4	0>
2	<8	0	0	0	0	0	0	-4	0	0	-4>
2	<8	0	0	0	0	4	0	4	0	0	0>
2	<8	0	0	0	0	0	-4	0	4	0	0>
2	<8	0	0	0	0	0	0	-4	0	-4	0>
2	<8	0	0	0	0	0	0	0	-4	0	-4>
2	<8	0	0	0	0	0	4	-4	0	0	0>
1	<8	0	0	0	0	4	-4	4	0	0	0>
1	<8	0	0	0	0	4	0	0	4	4	0>
1	<8	0	0	0	0	0	0	0	0	8	0>
1	<8	0	0	0	0	4	4	0	0	-4	4>
1	<8	0	0	0	0	0	-4	0	4	0	-4>
1	<8	0	0	0	0	4	0	-4	-4	0	-4>
1	<8	0	0	0	0	0	0	-4	0	4	4>
1	<8	0	0	0	0	0	0	0	0	-8	0>
1	<8	0	0	0	0	0	0	0	8	0	0>
1	<8	0	0	0	0	-4	0	0	4	4	0>
1	<8	0	0	0	0	0	0	-8	0	0	0>
1	<8	0	0	0	0	-4	4	0	0	-4	-4>
1	<8	0	0	0	0	4	-4	0	0	-4	-4>
1	<8	0	0	0	0	0	4	4	-4	4	0>
1	<8	0	0	0	0	0	0	0	-8	0	0>
1	<8	0	0	0	0	-4	-4	0	0	4	-4>
1	<8	0	0	0	0	0	0	4	0	4	-4>
1	<8	0	0	0	0	0	0	8	0	0	0>
1	<8	0	0	0	0	0	-4	4	-4	-4	0>
1	<8	0	0	0	0	-4	0	0	-4	-4	0>
1	<8	0	0	0	0	4	0	-4	4	0	4>
1	<8	0	0	0	0	0	-4	0	4	0	4>
1	<8	0	0	0	0	0	4	-4	4	4	0>
1	<8	0	0	0	0	-4	0	0	-4	4	0>
1	<8	0	0	0	0	-4	-4	0	0	-4	4>
1	<8	0	0	0	0	4	4	0	0	4	-4>
1	<8	0	0	0	0	4	0	0	4	-4	0>
1	<8	0	0	0	0	0	0	0	0	0	-8>
1	<8	0	0	0	0	8	0	0	0	0	0>
1	<8	0	0	0	0	4	4	-4	0	0	0>
1	<8	0	0	0	0	4	-4	-4	0	0	0>
1	<8	0	0	0	0	0	4	0	-4	0	4>
1	<8	0	0	0	0	-4	0	-4	-4	0	4>
1	<8	0	0	0	0	0	0	4	0	-4	-4>
1	<8	0	0	0	0	4	0	0	-4	4	0>
1	<8	0	0	0	0	-4	0	4	4	0	4>
1	<8	0	0	0	0	0	-8	0	0	0	0>
1	<8	0	0	0	0	0	-4	0	-4	0	4>
1	<8	0	0	0	0	0	4	4	4	-4	0>
1	<8	0	0	0	0	0	-4	-4	-4	4	0>
1	<8	0	0	0	0	-4	-4	-4	0	0	0>
1	<8	0	0	0	0	0	0	4	0	4	4>
1	<8	0	0	0	0	-4	-4	4	0	0	0>
1	<8	0	0	0	0	4	0	4	-4	0	4>
1	<8	0	0	0	0	4	0	0	-4	-4	0>
1	<8	0	0	0	0	-4	4	4	0	0	0>
1	<8	0	0	0	0	0	-4	0	-4	0	-4>

1	<8	0	0	0	0	−4	0	−4	4	0	−4>
1	<8	0	0	0	0	−4	4	0	0	4	4>
1	<8	0	0	0	0	0	0	4	0	−4	4>
1	<8	0	0	0	0	0	−4	4	4	4	0>
1	<8	0	0	0	0	−4	0	0	4	−4	0>
1	<8	0	0	0	0	0	8	0	0	0	0>
1	<8	0	0	0	0	0	4	0	4	0	−4>
1	<8	0	0	0	0	0	0	0	0	0	8>
1	<8	0	0	0	0	0	0	−4	0	−4	−4>
1	<8	0	0	0	0	4	−4	0	0	4	4>
1	<8	0	0	0	0	4	4	4	0	0	0>
1	<8	0	0	0	0	0	−4	−4	4	−4	0>
1	<8	0	0	0	0	0	0	−4	0	−4	4>
1	<8	0	0	0	0	0	4	−4	−4	−4	0>
1	<8	0	0	0	0	−4	4	−4	0	0	0>
1	<8	0	0	0	0	−4	0	4	−4	0	−4>
1	<8	0	0	0	0	−8	0	0	0	0	0>
1	<8	0	0	0	0	4	0	4	4	0	−4>
1	<8	0	0	0	0	0	0	−4	0	4	−4>
1	<8	0	0	0	0	0	4	0	−4	0	−4>
1	<8	0	0	0	0	0	4	0	4	0	4>

En total tenemos 129 clases de equivalencia que se corresponden con 222 funciones Booleanas.

Tenemos 1 clase de equivalencia que contiene 10 funciones Booleanas, 12 clases de equivalencia que contienen 4 funciones Booleanas cada una, 48 clases de equivalencia que cada una contienen 2 funciones Booleanas cada una y por último tenemos 68 clases que cada una contiene 1 función Booleana.

N4 K2 W8 ext 2

Podemos ver que estamos en el caso A ya que se cumple la siguiente igualdad:

$$\frac{8}{2^2} = 2 = 2^{4-2-1}$$

Cantidad funciones	Clase										
10	<8	0	0	0	0	0	0	0	0	0	0>

Tenemos una clase de equivalencia que se corresponde con 10 funciones Booleanas.

N4 K2 W8 ext 3

Al extender la clase de equivalencia a k+1 (3) se obtienen las siguientes clases de equivalencia:

Cantidad funciones	Clases														
2	<8	0	0	0	0	0	0	0	0	0	0	0	0	0	0>
1	<8	0	0	0	0	0	0	0	0	0	0	0	0	0	−8>
1	<8	0	0	0	0	0	0	0	0	0	0	−8	0	0	0>
1	<8	0	0	0	0	0	0	0	0	0	0	0	8	0	0>
1	<8	0	0	0	0	0	0	0	0	0	0	0	0	−8	0>
1	<8	0	0	0	0	0	0	0	0	0	0	0	−8	0	0>
1	<8	0	0	0	0	0	0	0	0	0	0	0	0	0	8>
1	<8	0	0	0	0	0	0	0	0	0	0	8	0	0	0>
1	<8	0	0	0	0	0	0	0	0	0	0	0	0	8	0>

En total tenemos 9 clases de equivalencia que se corresponden con 10 funciones Booleanas.

Tenemos una clase de equivalencia que se corresponde con 2 funciones Booleanas y tenemos 8 clases de equivalencia que cada una contiene una función Booleana.

N5 K2 W16 ext 2

Podemos ver que estamos en el caso A ya que se cumple la siguiente igualdad:

$$\frac{16}{2^2} = 4 = 2^{5-2-1}$$

Cantidad funciones	Clase															
552	<16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0>

Tenemos una clase de equivalencia que contiene 552 funciones Booleanas.

N5 K2 W16 ext 3

Al extender la clase de equivalencia a k+1 (3) se obtienen las siguientes clases de equivalencia:

Cantidad funciones	Clases											
12	<16	{15 0's}	0	0	0	0	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	0	0	0	0	8	0	0	0>
4	<16	{15 0's}	0	0	0	0	8	0	0	0	0	0>
4	<16	{15 0's}	-8	0	0	0	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	0	8	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	0	0	0	0	0	8	0	0>
4	<16	{15 0's}	0	0	0	0	0	0	-8	0	0	0>
4	<16	{15 0's}	0	0	0	0	0	8	0	0	0	0>
4	<16	{15 0's}	0	-8	0	0	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	0	0	0	0	0	0	-8	0>
4	<16	{15 0's}	0	0	0	0	0	0	0	0	0	-8>
4	<16	{15 0's}	0	0	0	0	0	0	0	-8	0	0>
4	<16	{15 0's}	0	0	-8	0	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	0	0	0	0	0	0	0	8>
4	<16	{15 0's}	0	8	0	0	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	8	0	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	0	0	0	-8	0	0	0	0>
4	<16	{15 0's}	8	0	0	0	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	0	0	0	0	0	0	8	0>
4	<16	{15 0's}	0	0	0	-8	0	0	0	0	0	0>
4	<16	{15 0's}	0	0	0	0	-8	0	0	0	0	0>
2	<16	{15 0's}	8	0	0	0	0	0	8	0	0	0>
2	<16	{15 0's}	8	-8	0	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	8	0	0	0	0	0	0	8	0>
2	<16	{15 0's}	0	0	8	0	0	-8	0	0	0	0>
2	<16	{15 0's}	0	0	0	-8	0	0	0	0	0	8>
2	<16	{15 0's}	-8	0	0	0	0	0	8	0	0	0>
2	<16	{15 0's}	-8	0	0	0	0	0	0	8	0	0>
2	<16	{15 0's}	8	0	0	-8	0	0	0	0	0	0>
2	<16	{15 0's}	8	0	0	0	0	0	-8	0	0	0>
2	<16	{15 0's}	0	0	0	8	0	0	8	0	0	0>
2	<16	{15 0's}	0	0	0	-8	0	8	0	0	0	0>
2	<16	{15 0's}	0	0	8	0	0	0	0	-8	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	-8	0	0	8>
2	<16	{15 0's}	0	0	0	0	0	0	-8	0	0	-8>
2	<16	{15 0's}	0	0	-8	0	0	0	0	0	-8	0>
2	<16	{15 0's}	0	0	0	0	8	0	0	8	0	0>
2	<16	{15 0's}	0	0	0	0	-8	0	0	0	0	-8>
2	<16	{15 0's}	0	-8	-8	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	8	0	0	0	0	0	-8>
2	<16	{15 0's}	-8	0	0	8	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	0	-8	-8>
2	<16	{15 0's}	0	8	0	0	0	0	0	0	-8	0>
2	<16	{15 0's}	0	0	0	0	0	-8	0	0	0	-8>
2	<16	{15 0's}	0	0	0	-8	0	0	0	0	0	-8>
2	<16	{15 0's}	0	-8	0	0	0	0	0	0	8	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	8	0	8>
2	<16	{15 0's}	0	0	0	0	0	0	8	0	0	8>
2	<16	{15 0's}	0	8	0	8	0	0	0	0	0	0>
2	<16	{15 0's}	8	0	0	0	-8	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	-8	8	0>
2	<16	{15 0's}	0	0	0	0	0	0	-8	8	0	0>

2	<16	{15 0's}	8	0	8	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	8	0	0	-8>
2	<16	{15 0's}	0	0	-8	0	0	-8	0	0	0	0>
2	<16	{15 0's}	0	0	-8	0	-8	0	0	0	0	0>
2	<16	{15 0's}	8	0	-8	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	0	-8	8>
2	<16	{15 0's}	0	0	0	0	0	-8	0	0	-8	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	8	8	0>
2	<16	{15 0's}	-8	0	-8	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	-8	0	0	0	8	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	-8	8	0	0	0	0>
2	<16	{15 0's}	0	-8	0	8	0	0	0	0	0	0>
2	<16	{15 0's}	0	8	0	0	0	-8	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	8	-8	0>
2	<16	{15 0's}	0	8	-8	0	0	0	0	0	0	0>
2	<16	{15 0's}	8	0	0	8	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	-8	8	0	0	0	0	0>
2	<16	{15 0's}	8	0	0	0	0	0	0	-8	0	0>
2	<16	{15 0's}	0	0	0	0	-8	0	0	-8	0	0>
2	<16	{15 0's}	0	0	0	0	8	0	0	-8	0	0>
2	<16	{15 0's}	0	0	0	0	0	-8	0	0	8	0>
2	<16	{15 0's}	0	0	-8	0	0	8	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	0	8	-8>
2	<16	{15 0's}	0	-8	0	-8	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	-8	-8	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	8	0	-8>
2	<16	{15 0's}	0	0	0	8	0	0	-8	0	0	0>
2	<16	{15 0's}	0	0	0	8	8	0	0	0	0	0>
2	<16	{15 0's}	-8	0	0	0	0	0	-8	0	0	0>
2	<16	{15 0's}	0	8	0	0	0	8	0	0	0	0>
2	<16	{15 0's}	0	0	0	8	0	-8	0	0	0	0>
2	<16	{15 0's}	0	0	0	8	-8	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	8	8	0	0>
2	<16	{15 0's}	0	0	8	0	0	0	0	8	0	0>
2	<16	{15 0's}	0	-8	0	0	0	0	-8	0	0	0>
2	<16	{15 0's}	0	8	0	0	0	0	8	0	0	0>
2	<16	{15 0's}	-8	0	0	0	-8	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	8	0	8	0>
2	<16	{15 0's}	0	0	0	0	-8	0	0	8	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	-8	-8	0>
2	<16	{15 0's}	0	-8	0	0	0	-8	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	-8	0	8	0>
2	<16	{15 0's}	0	0	0	0	0	8	0	0	-8	0>
2	<16	{15 0's}	0	0	0	0	0	8	0	0	0	8>
2	<16	{15 0's}	0	0	0	0	0	0	-8	0	-8	0>
2	<16	{15 0's}	0	0	0	-8	0	-8	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	-8	0	0	0	0	8>
2	<16	{15 0's}	0	-8	8	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	0	0	8	8>
2	<16	{15 0's}	0	0	0	0	0	0	0	-8	0	-8>
2	<16	{15 0's}	-8	8	0	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	-8	-8	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	8	0	0	0	0	0	8>
2	<16	{15 0's}	0	0	0	0	8	-8	0	0	0	0>
2	<16	{15 0's}	0	8	0	-8	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	-8	0	0	0	0	8	0	0>
2	<16	{15 0's}	0	0	0	0	8	0	0	0	0	-8>
2	<16	{15 0's}	0	0	0	0	0	0	0	-8	0	8>
2	<16	{15 0's}	0	0	0	0	8	8	0	0	0	0>
2	<16	{15 0's}	0	0	8	0	0	8	0	0	0	0>
2	<16	{15 0's}	0	0	0	-8	0	0	8	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	-8	-8	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	8	0	-8	0>
2	<16	{15 0's}	-8	0	8	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	8	0	0	0	0	0	-8	0>
2	<16	{15 0's}	0	0	8	0	8	0	0	0	0	0>
2	<16	{15 0's}	0	-8	0	0	0	0	0	0	-8	0>
2	<16	{15 0's}	8	0	0	0	8	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	8	0	0	0	-8>
2	<16	{15 0's}	-8	0	0	0	0	0	0	0	-8	0>
2	<16	{15 0's}	0	8	0	0	0	0	-8	0	0	0>
2	<16	{15 0's}	-8	0	0	-8	0	0	0	0	0	0>

2	<16	{15 0's}	0	0	0	0	8	0	0	0	0	8>
2	<16	{15 0's}	0	0	0	-8	0	0	-8	0	0	0>
2	<16	{15 0's}	0	0	0	8	0	8	0	0	0	0>
2	<16	{15 0's}	-8	0	0	0	8	0	0	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	-8	0	0	0	8>
2	<16	{15 0's}	8	0	0	0	0	0	0	8	0	0>
2	<16	{15 0's}	0	8	8	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	-8	0	0	0	0	-8	0	0>
2	<16	{15 0's}	0	0	0	0	0	8	0	0	8	0>
2	<16	{15 0's}	-8	-8	0	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	0	8	0	-8	0	0	0	0	0>
2	<16	{15 0's}	0	0	-8	0	0	0	0	0	8	0>
2	<16	{15 0's}	0	0	8	0	0	0	0	0	8	0>
2	<16	{15 0's}	8	8	0	0	0	0	0	0	0	0>
2	<16	{15 0's}	0	-8	0	0	0	0	8	0	0	0>
2	<16	{15 0's}	0	0	0	0	0	0	8	-8	0	0>
2	<16	{15 0's}	0	0	-8	0	8	0	0	0	0	0>
1	<16	{15 0's}	-8	0	0	0	-8	0	8	0	0	-8>
1	<16	{15 0's}	0	8	0	0	0	0	-8	0	8	0>
1	<16	{15 0's}	0	0	-8	0	0	0	0	-8	-8	0>
1	<16	{15 0's}	0	0	0	0	-8	0	0	8	0	8>
1	<16	{15 0's}	-8	8	0	0	-8	-8	0	0	0	0>
1	<16	{15 0's}	0	8	0	0	0	-8	-8	0	0	-8>
1	<16	{15 0's}	8	0	0	-8	-8	0	0	0	0	0>
1	<16	{15 0's}	0	0	0	-8	0	0	-8	0	0	8>
1	<16	{15 0's}	8	0	0	8	0	0	0	8	0	-8>
1	<16	{15 0's}	0	0	0	-8	0	8	8	0	8	0>
1	<16	{15 0's}	0	0	8	0	8	0	0	0	-8	8>
1	<16	{15 0's}	8	-8	0	0	0	0	0	8	8	0>
1	<16	{15 0's}	0	0	0	8	0	8	-8	0	8	0>
1	<16	{15 0's}	-16	0	0	0	0	0	0	0	0	0>
1	<16	{15 0's}	0	8	-8	0	0	0	-8	-8	0	0>
1	<16	{15 0's}	-8	0	-8	8	0	-8	0	0	0	0>
1	<16	{15 0's}	0	0	-8	0	0	8	0	-8	0	-8>
1	<16	{15 0's}	0	-8	8	8	8	0	0	0	0	0>
1	<16	{15 0's}	0	0	8	0	0	0	0	8	8	0>
1	<16	{15 0's}	0	0	0	8	8	0	8	-8	0	0>
1	<16	{15 0's}	0	0	0	-8	0	0	8	0	0	-8>
1	<16	{15 0's}	0	0	0	0	0	8	0	0	-8	8>
1	<16	{15 0's}	8	8	8	0	0	0	0	0	0	0>
1	<16	{15 0's}	0	0	0	-8	0	-8	8	0	-8	0>
1	<16	{15 0's}	0	8	0	-8	0	8	0	0	0	0>
1	<16	{15 0's}	8	0	0	8	8	0	0	0	0	0>
1	<16	{15 0's}	0	0	8	0	0	0	0	-8	8	0>
1	<16	{15 0's}	-8	0	0	0	0	0	-8	8	0	0>
1	<16	{15 0's}	0	-8	0	8	0	8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	0	8	0	0	8	-8>
1	<16	{15 0's}	0	0	0	0	0	8	0	0	-8	-8>
1	<16	{15 0's}	0	0	0	0	0	-8	0	0	-8	-8>
1	<16	{15 0's}	0	8	8	0	0	0	8	-8	0	0>
1	<16	{15 0's}	-8	0	0	-8	0	0	0	8	0	-8>
1	<16	{15 0's}	8	0	0	8	-8	0	0	0	0	0>
1	<16	{15 0's}	8	0	0	0	0	0	8	8	0	0>
1	<16	{15 0's}	0	0	0	0	0	0	0	0	0	-16>
1	<16	{15 0's}	0	-8	0	0	0	0	-8	0	-8	0>
1	<16	{15 0's}	0	0	-8	0	0	-8	0	-8	0	8>
1	<16	{15 0's}	0	0	0	-8	-8	0	-8	8	0	0>
1	<16	{15 0's}	0	8	0	-8	0	-8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	8	0	0	8	0	-8>
1	<16	{15 0's}	-8	8	8	0	0	0	0	0	0	0>
1	<16	{15 0's}	8	0	0	0	0	0	-8	8	0	0>
1	<16	{15 0's}	0	0	8	0	8	0	0	0	8	-8>
1	<16	{15 0's}	0	0	0	8	0	0	8	0	0	8>
1	<16	{15 0's}	8	0	8	8	0	-8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	0	16	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	-8	0	0	-8	0	8>
1	<16	{15 0's}	0	0	-8	0	8	0	0	0	-8	-8>
1	<16	{15 0's}	0	0	0	0	0	0	0	0	16	0>
1	<16	{15 0's}	0	0	0	0	8	0	0	-8	0	8>
1	<16	{15 0's}	-8	0	8	0	0	0	-8	0	-8	0>
1	<16	{15 0's}	8	0	0	0	8	0	8	0	0	-8>
1	<16	{15 0's}	0	0	8	0	0	0	0	-8	-8	0>

1	<16	{15 0's}	0	-8	0	0	0	-8	8	0	0	-8>
1	<16	{15 0's}	0	0	0	-8	0	0	-8	0	0	-8>
1	<16	{15 0's}	-8	0	0	0	0	0	8	-8	0	0>
1	<16	{15 0's}	-8	0	-8	-8	0	8	0	0	0	0>
1	<16	{15 0's}	0	8	0	-8	0	0	0	0	8	8>
1	<16	{15 0's}	-8	-8	-8	0	0	0	0	0	0	0>
1	<16	{15 0's}	-8	0	0	8	0	0	0	8	0	8>
1	<16	{15 0's}	8	8	0	0	0	0	0	8	-8	0>
1	<16	{15 0's}	0	0	0	0	0	-8	0	0	8	8>
1	<16	{15 0's}	0	-16	0	0	0	0	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	0	0	0	16	0	0>
1	<16	{15 0's}	0	0	8	0	0	0	0	8	-8	0>
1	<16	{15 0's}	8	0	8	-8	0	8	0	0	0	0>
1	<16	{15 0's}	-8	0	8	0	0	0	8	0	8	0>
1	<16	{15 0's}	0	0	0	8	-8	0	8	8	0	0>
1	<16	{15 0's}	8	0	0	-8	8	0	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	8	0	0	8	0	8>
1	<16	{15 0's}	0	0	8	0	0	8	0	8	0	-8>
1	<16	{15 0's}	0	8	0	0	0	0	-8	0	-8	0>
1	<16	{15 0's}	0	8	-8	0	0	0	8	8	0	0>
1	<16	{15 0's}	0	0	0	0	-8	0	0	-8	0	-8>
1	<16	{15 0's}	0	-8	0	-8	0	8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	0	0	-16	0	0	0>
1	<16	{15 0's}	-8	0	0	0	0	0	8	8	0	0>
1	<16	{15 0's}	0	0	8	0	-8	0	0	0	-8	-8>
1	<16	{15 0's}	8	8	0	0	-8	8	0	0	0	0>
1	<16	{15 0's}	-8	0	0	0	0	0	-8	-8	0	0>
1	<16	{15 0's}	0	0	-8	0	0	0	0	-8	8	0>
1	<16	{15 0's}	-8	8	0	0	8	8	0	0	0	0>
1	<16	{15 0's}	8	0	-8	8	0	8	0	0	0	0>
1	<16	{15 0's}	0	-8	0	0	0	8	-8	0	0	-8>
1	<16	{15 0's}	8	0	-8	-8	0	-8	0	0	0	0>
1	<16	{15 0's}	0	0	8	0	-8	8	0	0	0	0>
1	<16	{15 0's}	-8	-8	0	0	0	0	0	8	-8	0>
1	<16	{15 0's}	0	16	0	0	0	0	0	0	0	0>
1	<16	{15 0's}	-8	8	-8	0	0	0	0	0	0	0>
1	<16	{15 0's}	8	0	-8	0	0	0	-8	0	-8	0>
1	<16	{15 0's}	0	0	-8	0	-8	8	0	0	0	0>
1	<16	{15 0's}	0	0	0	8	0	0	-8	0	0	8>
1	<16	{15 0's}	0	0	0	8	8	0	-8	8	0	0>
1	<16	{15 0's}	-8	0	0	0	-8	0	-8	0	0	8>
1	<16	{15 0's}	0	0	0	-8	8	0	8	8	0	0>
1	<16	{15 0's}	0	0	8	0	8	8	0	0	0	0>
1	<16	{15 0's}	-8	-8	0	0	0	0	0	-8	8	0>
1	<16	{15 0's}	0	-8	0	-8	0	-8	0	0	0	0>
1	<16	{15 0's}	0	0	-8	0	-8	0	0	0	-8	8>
1	<16	{15 0's}	-8	0	0	-8	0	0	0	-8	0	8>
1	<16	{15 0's}	0	0	8	0	8	-8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	-8	-8	0	-8	8	0>
1	<16	{15 0's}	0	-8	-8	-8	8	0	0	0	0	0>
1	<16	{15 0's}	0	8	-8	8	8	0	0	0	0	0>
1	<16	{15 0's}	0	8	-8	-8	-8	0	0	0	0	0>
1	<16	{15 0's}	0	0	0	-8	8	0	-8	-8	0	0>
1	<16	{15 0's}	0	8	0	8	0	0	0	0	-8	8>
1	<16	{15 0's}	8	0	0	0	-8	0	8	0	0	8>
1	<16	{15 0's}	-8	0	0	-8	8	0	0	0	0	0>
1	<16	{15 0's}	0	-8	-8	8	-8	0	0	0	0	0>
1	<16	{15 0's}	0	-8	-8	0	0	0	-8	8	0	0>
1	<16	{15 0's}	-8	0	8	8	0	8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	8	8	0	-8	8	0>
1	<16	{15 0's}	8	0	0	0	0	0	8	-8	0	0>
1	<16	{15 0's}	0	8	0	-8	0	0	0	0	-8	-8>
1	<16	{15 0's}	0	0	-8	0	0	-8	0	8	0	-8>
1	<16	{15 0's}	0	0	8	0	0	-8	0	-8	0	-8>
1	<16	{15 0's}	0	0	8	0	0	8	0	-8	0	8>
1	<16	{15 0's}	0	0	0	0	8	0	0	-8	0	-8>
1	<16	{15 0's}	0	8	0	0	0	0	8	0	-8	0>
1	<16	{15 0's}	0	0	0	0	0	8	0	0	8	8>
1	<16	{15 0's}	0	-8	0	8	0	0	0	0	8	8>
1	<16	{15 0's}	0	-8	0	-8	0	0	0	0	-8	8>
1	<16	{15 0's}	0	0	0	8	-8	0	-8	-8	0	0>
1	<16	{15 0's}	0	0	0	0	0	0	16	0	0	0>

1	<16	{15 0's}	-8	-8	0	0	8	-8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	-8	8	0	8	8	0>
1	<16	{15 0's}	0	0	-8	0	0	0	0	8	-8	0>
1	<16	{15 0's}	0	0	0	8	0	0	-8	0	0	-8>
1	<16	{15 0's}	8	0	0	0	8	0	-8	0	0	8>
1	<16	{15 0's}	0	-8	-8	0	0	0	8	-8	0	0>
1	<16	{15 0's}	0	0	0	0	0	-16	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	0	0	0	0	0	16>
1	<16	{15 0's}	0	-8	0	0	0	8	8	0	0	8>
1	<16	{15 0's}	0	-8	0	0	0	0	8	0	-8	0>
1	<16	{15 0's}	8	8	0	0	8	-8	0	0	0	0>
1	<16	{15 0's}	0	8	0	8	0	0	0	0	8	-8>
1	<16	{15 0's}	8	0	0	0	0	0	-8	-8	0	0>
1	<16	{15 0's}	0	0	-8	0	8	8	0	0	0	0>
1	<16	{15 0's}	8	0	0	8	0	0	0	-8	0	8>
1	<16	{15 0's}	0	0	0	-8	0	8	-8	0	-8	0>
1	<16	{15 0's}	8	-8	0	0	-8	-8	0	0	0	0>
1	<16	{15 0's}	0	0	0	-8	0	0	8	0	0	8>
1	<16	{15 0's}	-8	0	0	8	0	0	0	-8	0	-8>
1	<16	{15 0's}	8	0	0	-8	0	0	0	8	0	8>
1	<16	{15 0's}	0	-8	0	0	0	0	-8	0	8	0>
1	<16	{15 0's}	0	8	0	0	0	0	8	0	8	0>
1	<16	{15 0's}	0	-8	8	0	0	0	-8	-8	0	0>
1	<16	{15 0's}	0	8	0	8	0	8	0	0	0	0>
1	<16	{15 0's}	-8	8	0	0	0	0	0	8	8	0>
1	<16	{15 0's}	0	0	0	-16	0	0	0	0	0	0>
1	<16	{15 0's}	0	0	8	0	-8	-8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	16	0	0	0	0	0>
1	<16	{15 0's}	0	-8	0	8	0	0	0	0	-8	-8>
1	<16	{15 0's}	-8	8	0	0	0	0	0	-8	-8	0>
1	<16	{15 0's}	16	0	0	0	0	0	0	0	0	0>
1	<16	{15 0's}	-8	-8	0	0	-8	8	0	0	0	0>
1	<16	{15 0's}	0	0	0	8	0	-8	8	0	8	0>
1	<16	{15 0's}	0	0	0	16	0	0	0	0	0	0>
1	<16	{15 0's}	0	0	-8	0	8	-8	0	0	0	0>
1	<16	{15 0's}	0	-8	0	-8	0	0	0	0	8	-8>
1	<16	{15 0's}	0	0	0	0	0	0	0	0	-16	0>
1	<16	{15 0's}	-8	-8	8	0	0	0	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	0	0	0	-16	0	0>
1	<16	{15 0's}	0	0	0	8	0	-8	-8	0	-8	0>
1	<16	{15 0's}	0	-8	8	0	0	0	8	8	0	0>
1	<16	{15 0's}	0	0	0	0	8	8	0	8	-8	0>
1	<16	{15 0's}	8	-8	0	0	0	0	0	-8	-8	0>
1	<16	{15 0's}	0	-8	0	0	0	-8	-8	0	0	8>
1	<16	{15 0's}	-8	0	8	-8	0	-8	0	0	0	0>
1	<16	{15 0's}	-8	0	0	0	8	0	-8	0	0	-8>
1	<16	{15 0's}	0	8	0	0	0	8	-8	0	0	8>
1	<16	{15 0's}	0	0	-8	0	-8	0	0	0	8	-8>
1	<16	{15 0's}	0	0	0	0	-8	8	0	-8	-8	0>
1	<16	{15 0's}	8	0	0	-8	0	0	0	-8	0	-8>
1	<16	{15 0's}	0	0	0	-8	-8	0	8	-8	0	0>
1	<16	{15 0's}	0	0	0	8	0	0	8	0	0	-8>
1	<16	{15 0's}	0	0	-16	0	0	0	0	0	0	0>
1	<16	{15 0's}	-8	0	0	8	8	0	0	0	0	0>
1	<16	{15 0's}	8	8	-8	0	0	0	0	0	0	0>
1	<16	{15 0's}	8	8	0	0	0	0	0	-8	8	0>
1	<16	{15 0's}	0	0	16	0	0	0	0	0	0	0>
1	<16	{15 0's}	8	0	8	0	0	0	-8	0	8	0>
1	<16	{15 0's}	0	0	0	0	-8	-8	0	8	-8	0>
1	<16	{15 0's}	0	0	-8	0	8	0	0	0	8	8>
1	<16	{15 0's}	0	0	-8	0	0	0	0	8	8	0>
1	<16	{15 0's}	0	0	0	0	0	-8	0	0	-8	8>
1	<16	{15 0's}	-8	0	0	-8	-8	0	0	0	0	0>
1	<16	{15 0's}	8	-8	8	0	0	0	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	8	-8	0	8	8	0>
1	<16	{15 0's}	0	-8	0	8	0	-8	0	0	0	0>
1	<16	{15 0's}	0	8	8	8	-8	0	0	0	0	0>
1	<16	{15 0's}	0	8	8	-8	8	0	0	0	0	0>
1	<16	{15 0's}	0	8	0	0	0	-8	8	0	0	8>
1	<16	{15 0's}	0	0	-8	0	0	8	0	8	0	8>
1	<16	{15 0's}	8	0	0	0	-8	0	-8	0	0	-8>
1	<16	{15 0's}	-8	0	-8	0	0	0	-8	0	8	0>

1	<16	{15 0's}	0	8	8	0	0	0	-8	8	0	0>
1	<16	{15 0's}	0	0	-8	0	-8	-8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	8	-8	0	-8	-8	0>
1	<16	{15 0's}	0	0	0	8	0	8	8	0	-8	0>
1	<16	{15 0's}	0	0	8	0	0	-8	0	8	0	8>
1	<16	{15 0's}	-8	0	0	0	8	0	8	0	0	8>
1	<16	{15 0's}	8	-8	0	0	8	8	0	0	0	0>
1	<16	{15 0's}	8	-8	-8	0	0	0	0	0	0	0>
1	<16	{15 0's}	8	0	8	0	0	0	8	0	-8	0>
1	<16	{15 0's}	0	8	0	0	0	8	8	0	0	-8>
1	<16	{15 0's}	0	-8	0	0	0	0	8	0	8	0>
1	<16	{15 0's}	0	0	0	0	0	-8	0	0	8	-8>
1	<16	{15 0's}	0	8	0	8	0	-8	0	0	0	0>
1	<16	{15 0's}	0	0	0	0	-16	0	0	0	0	0>
1	<16	{15 0's}	-8	0	-8	0	0	0	8	0	-8	0>
1	<16	{15 0's}	0	0	8	0	-8	0	0	0	8	8>
1	<16	{15 0's}	0	0	0	0	-8	0	0	8	0	-8>
1	<16	{15 0's}	0	0	0	-8	0	-8	-8	0	8	0>
1	<16	{15 0's}	0	-8	8	-8	-8	0	0	0	0	0>
1	<16	{15 0's}	8	0	-8	0	0	0	8	0	8	0>
1	<16	{15 0's}	-8	0	0	8	-8	0	0	0	0	0>

En total tenemos 361 clases de equivalencia que contienen 552 funciones Booleanas.

Tenemos una clase de equivalencia que contiene 12 funciones Booleanas, también tenemos 20 clases de equivalencia que cada una contiene 4 funciones Booleanas. Tenemos 120 clases de equivalencia que cada una contiene 2 funciones Booleanas y por último tenemos 220 clases de equivalencia que cada una contiene una función Booleana.

N5 K3 W16 ext 3

Podemos ver que estamos en el caso A ya que se cumple la siguiente igualdad:

$$\frac{16}{2^3} = 2 = 2^{5-3-1}$$

Cantidad funciones	Clase	
12	<16	{25 0's}>

Tenemos una clase de equivalencia que contiene 12 funciones Booleanas.

N5 K3 W16 ext 4

Al extender la clase de equivalencia a k+1 (4) se obtienen las siguientes clases de equivalencia:

Cantidad funciones	Clases						
2	<16	{25 0's}	0	0	0	0	0>
1	<16	{25 0's}	0	-16	0	0	0>
1	<16	{25 0's}	0	0	-16	0	0>
1	<16	{25 0's}	0	0	0	0	-16>
1	<16	{25 0's}	0	16	0	0	0>
1	<16	{25 0's}	-16	0	0	0	0>
1	<16	{25 0's}	0	0	0	0	16>
1	<16	{25 0's}	0	0	0	-16	0>
1	<16	{25 0's}	16	0	0	0	0>
1	<16	{25 0's}	0	0	16	0	0>
1	<16	{25 0's}	0	0	0	16	0>

En total tenemos 11 clases que contienen 12 funciones Booleanas.

Tenemos una clase de equivalencia que contiene 2 funciones Booleanas y tenemos 10 clases de equivalencia que cada una contiene una función Booleana.

Caso B

En los casos que no cumplen con la propiedad planteada en el Caso A, se notó que la cantidad de clases resultantes en la extensión se correspondía con la cantidad de funciones.

Este comportamiento impacta fuertemente cuando se quiere generar nuevas clases a partir de predecesoras ya que al extender se estará trabajando con una clase por función, es decir todas las funciones Booleanas para ese N y K.

A continuación, mostraremos la extensión de ejemplo de este caso:

N4 K1 W8 ext 1

Cantidad de clases inmunes a la correlación: 1

Cantidad de funciones: 8

Cantidad funciones	Clase				
8	<2	0	0	0	0>

N4 K1 W8 ext 2

Cantidad de clases inmunes a la correlación: 8

Cantidad de funciones: 8

Cantidad funciones	Clases										
1	<2	0	0	0	0	2	2	2	2	2	2>
1	<2	0	0	0	0	-2	2	-2	-2	2	-2>
1	<2	0	0	0	0	-2	-2	2	2	-2	-2>
1	<2	0	0	0	0	2	-2	-2	-2	-2	2>
1	<2	0	0	0	0	2	2	-2	2	-2	-2>
1	<2	0	0	0	0	2	-2	2	-2	2	-2>
1	<2	0	0	0	0	-2	-2	-2	2	2	2>
1	<2	0	0	0	0	-2	2	2	-2	-2	2>

Como un trabajo a futuro se pueden analizar los resultados obtenidos para todos los casos ejecutados, ya que se pudieron observar algunos comportamientos de patrones en las clases resultado de la extensión a un grado más.

ANEXO V - DATOS SOBRE GESTIÓN DEL PROYECTO

El proyecto se inició en marzo del 2014 y se extendió hasta julio del 2015.

Las tareas llevadas a cabo durante el proyecto fueron:

- Estudio de definiciones relacionadas a la temática del proyecto y trabajos de otros autores. Duración aprox: 2 meses
- Realización de cálculos manuales y elaboración en papel de algoritmo para la generación de clases. Duración aprox: 1 mes
- Evaluación de lenguaje, software de base y hardware. Pruebas. Duración aprox: 1 mes
- Diseño de estructura de datos y diseño de arquitectura de la aplicación: Duración aprox: 1 mes
- Implementación parte I, ejecución y pruebas. Duración aprox: 2 meses
- Modificación de estructura de datos parte II para mayor eficiencia Duración aprox: 1 mes
- Implementación parte II, ejecución y pruebas. Duración aprox: 2 mes
- Elaboración del informe del proyecto. Duración aprox: 5 meses

Desvios: Periodos de exámenes.

La cantidad de horas dedicadas al proyecto en promedio fue de 4 a 5 diarias, y entre 20 y 30hs semanales.

Al inicio del proyecto fue necesario realizar reuniones con mucha frecuencia con Alfredo para poder entender la problemática a resolver en el proyecto, así como también para estudiar con su ayuda todo el material teórico relacionado. Luego de tener una implementación las reuniones pasaron a ser mensuales para realizar, junto a él, el seguimiento del proyecto y discutir sobre resultados y mejoras posibles.