Minimización de Funciones Lógicas

El algoritmo de Quine – McClusky explicado y mejorado

Macluskey, 2023

ABSTRACT

El algoritmo de Quine-McClusky está considerado como el algoritmo de referencia para la minimización de funciones lógicas, es decir, aquellas que constan de un conjunto de variables booleanas unidas entre sí por los dos operadores lógicos: suma y producto, problema que está computacionalmente catalogado como *NP-completo*.

Dicho algoritmo asegura que, dada una determinada función lógica, obtiene siempre la expresión lógica mínima equivalente como Suma de Productos, es decir, con su misma tabla de verdad, pero con el menor número posible de términos. Una suma de productos puede siempre ser reducida a otra expresión con un menor número de variables individuales aplicando la propiedad distributiva para extraer los posibles factores comunes de los diferentes términos; más allá de esta transformación obvia, el algoritmo QM está reputado como la solución definitiva para minimizar una expresión lógica.

En este documento demostraré que en un importante porcentaje de casos, cercano al 50% en cuanto el tamaño de la forma canónica crece por encima de un cierto valor, el algoritmo no devuelve la expresión mínima equivalente, que sin embargo sí puede ser obtenida modificando en cierta medida el propio algoritmo.

Adicionalmente, se proponen una serie de modificaciones al algoritmo, incluyendo o modificando algunos de sus pasos para reducir en lo posible el consumo de recursos necesarios para su ejecución. En concreto, los puntos más importantes expresados en el documento son:

a) Inclusión de un nuevo Paso 1 - Detección y eliminación de variables implicantes que, dependiendo de las condiciones de la expresión tratada, puede reducir la necesidad de recursos informáticos necesarios para la obtención de la función mínima a la raíz cuadrada de los utilizados si no existiera este Paso 1.
b) Explicación de por qué el algoritmo no devuelve la expresión mínima absoluta en un importante número de casos, y cómo puede obtenerse dicha expresión mínima tratando los *maxterms* y complementando el resultado mediante las Leyes de De Morgan.
c) Demostración de que el número de expresiones en que la función mínima se obtiene tratando los *minterms* es el mismo que el número de expresiones en que la función mínima se obtiene tratando los *maxterms* y complementando la función obtenida.
d) Se demuestra que, computacionalmente, la minimización de funciones lógicas y el problema de cobertura de conjuntos (*Set Cover Problem*) es exactamente el mismo problema y, por lo tanto, todos los métodos y algoritmos utilizados para tratar uno de estos problemas son directamente aplicables a tratar el otro.
e) Se define detalladamente el Método de Petrick usado en el Paso 7 del algoritmo QM, proponiendo un algoritmo muy eficaz para resolverlo.
f) Se proponen modificaciones al Algoritmo Voraz (*Greedy Algorithm*) para su uso en el Paso 7 del algoritmo QM. El algoritmo propuesto tiene una tasa de acierto elevadísima comparada con la del algoritmo original, es decir, encuentra la función mínima absoluta en porcentajes superiores al 99% para las formas canónicas de hasta 256 bits (ocho variables individuales), corriendo siempre en tiempo polinómico.

Para confeccionar el documento se han realizado muchos millones de ejecuciones del algoritmo con diferentes tamaños de forma canónica y distintas codificaciones de los diferentes pasos del algoritmo; todos los datos que se muestran son reales, el resultado de dichas ejecuciones.

Introducción

El algoritmo de Quine-McClusky es conocido como el standard unánimemente aceptado tanto en informática como en electrónica para minimizar funciones lógicas, es decir, dada una determinada expresión booleana compuesta de varias condiciones individuales conectadas por los operadores lógicos + (OR) y * (AND) el algoritmo encuentra una expresión equivalente (con la misma tabla de verdad que la función dada) que es mínima, en el sentido de que tiene el menor número posible de condiciones individuales y operadores lógicos.

Entonces, ¿por qué dedico este extenso documento a explicar, desmenuzar, criticar e incluso proponer modificaciones a un algoritmo tan bien conocido y establecido en la profesión? Pues porque la descripción del algoritmo que se encuentra en la Red es en la mayor parte de las ocasiones difusa, incompleta y, en ocasiones, incluso errónea. Uno de mis objetivos es explicar detalladamente cada paso del algoritmo tal como se describe en la literatura, para tener, por así decirlo, un lugar de referencia donde poder averiguar cómo implementar el algoritmo en un ordenador. Y, más importante aún, porque en los muchos meses de investigación sobre las múltiples sutilezas del algoritmo **he hallado formas de mejorarlo**, de hacerlo más eficiente o, simplemente, de hacer que cumpla su cometido de encontrar una función mínima siempre y en toda ocasión, cosa que no siempre hace.

No pretendo en absoluto que estos "descubrimientos" que he realizado sean primicias mundiales. Hay tantísimo trabajo realizado sobre este algoritmo en los casi 70 años transcurridos desde que fue publicado que me sorprendería mucho que esos hallazgos que yo he hecho no los haya encontrado hace muchos años un doctorando en Alabama, un profesional en Munich o un investigador en Vitoria... lo único que aquí digo es que tras buscar lo más exhaustivamente que he sabido en la Red, en varios idiomas y con diferentes buscadores, no he podido hallar referencia alguna sobre estos descubrimientos. Todos ellos los explicaré en este documento para que quien lo desee los use si le son de utilidad.

En enero de 2023 publiqué en ElCedazo, blog comunitario de ElTamiz, el blog científico de Pedro Gómez Esteban, una serie de cinco artículos denominada como este mismo documento: Minimización de Funciones Lógicas – El algoritmo de Quine-McClusky explicado y mejorado. La serie tiene un sexto artículo en el que se publica este documento, que es el compendio de los cinco artículos antedichos.

Antes de entrar en harina con el algoritmo de Quine-McClusky, una breve explicación de por qué mi inusitado interés, a estas alturas de mi vida, en este proceloso algoritmo.

La **optimización de programas fuente** siempre me ha interesado mucho a lo largo de mi carrera profesional. Me refiero a optimizar el código fuente de los programas, es decir, sustituir unas ciertas instrucciones por otras que permitan que el programa resultante sea más eficiente y que se entendiera mejor de cara a su explotación y al inevitable mantenimiento posterior.

Uno de los puntos clave para optimizar el código fuente es, desde luego, la minimización de funciones lógicas, aquellas condiciones que el programador ha escrito para dirigir el flujo del programa, así que este tema es uno de los más importantes a tener en cuenta para optimizar el código fuente: sustituir en él una condición compleja por otra que, teniendo la misma funcionalidad, es decir, la misma tabla de verdad, sea más corta, lo más corta posible. Y no sólo a mí me interesa la minimización de funciones lógicas; se trata de un problema muy estudiado desde el mismísimo principio de la profesión. Entremos un poco más en detalle.

La minimización de funciones lógicas (o booleanas, es decir, aquellas cuyas variables sólo pueden tener dos valores: 0 - falso y 1 – verdadero), del estilo de "*Fecha mayor que 2022-01-01*" o "*Importe menor que Saldo"*, por ejemplo, es un problema común cuando se trata de optimizar un circuito o un programa que tiene que lidiar con dichas funciones lógicas. Cuantas menos variables tenga dicha función más sencillo será construir el circuito, y menos tiempo se requerirá para evaluar dicha función para cada juego de valores que puedan adoptar las variables individuales.

Pero hay un inconveniente: este problema es lo que en teoría de computación se denomina un *problema NP-Completo*. Resoluble, sí, pero a costa de consumir una cantidad indecente de recursos informáticos, memoria y tiempo de CPU, en cuanto el número de variables individuales crece por encima de un cierto punto, diez o doce, quizás quince. Conocido esto, desde los comienzos de la informática y la electrónica se trató de encontrar un método simple y eficaz que, siempre y en toda ocasión, obtenga una función equivalente a la función dada y que ésta sea mínima. Desde hace ya muchos años los informáticos disponemos de dos algoritmos para minimizar funciones lógicas: el de Karnaugh y el de Quine-McClusky.

El **método de Karnaugh** es un método fundamentalmente visual y está indicado cuando las funciones no tienen más allá de cuatro o, como máximo, cinco variables individuales. Nuestro amigo y autor J nos explicó este método con su habitual maestría hace un tiempo. Es factible escribir un programa para implementarlo, pero lo que para un ojo avezado resulta evidente no lo es tanto para un ordenador, y por eso nadie que yo sepa usa este método en cuanto el número de variables aumenta. Para eso está el segundo de los algoritmos citados: el de Quine-McClusky, al que no le importa el número de variables de la función a minimizar... siempre que se disponga de la memoria y del tiempo de CPU suficiente.

En cuanto al **algoritmo de Quine-McClusky**, es muy conocido en la industria informática y electrónica. Data de 1952, cuando Willard V. Quine publicó un artículo sobre "El problema de simplificar funciones de verdad", que fue perfeccionado en 1956 por Edward J. McClusky cuando publicó su tesis "Minimización de funciones lógicas", en la que definió el algoritmo definitivo de minimización de funciones lógicas.

En la Red se encuentra mucha documentación sobre este algoritmo, vídeos, tesis doctorales, ejemplos en los que se explica con mejor o peor suerte los pasos de que consta, incluso hay alguna página web que lo aplica a ciertos datos que el usuario proporciona y lo resuelve online, está en la práctica totalidad de los currícula de las Escuelas de Ingeniería Informática y Electrónica, siempre haciendo hincapié en el hecho de que el algoritmo, siempre y en toda ocasión, proporciona la función mínima correspondiente a una determinada forma canónica, es decir, a la tabla de verdad de la función dada: al algoritmo QM le importa poco conocer de dónde viene dicha tabla de verdad, pues ésa, la forma canónica, es la única entrada que necesita.

En resumen, **el algoritmo de Quine-McClusky es el Santo Grial de la minimización de funciones lógicas**. Está establecido como tal en todas partes, todos los profesores de lógica informática lo enseñan, los doctorandos lo utilizan y nadie se cuestiona si siempre funciona o no. El único problema reconocido para su aplicación, claro está, es la ingente cantidad de recursos informáticos, CPU y memoria, que necesita en cuanto el número de variables crece, consecuencia obvia de tratarse de un problema NP-Completo. En los casos en que el consumo de recursos se vuelve inabordable casi todo el mundo utiliza Espresso, el programa basado en heurísticos inicialmente creado por IBM en los años 70, actualmente disponible en las librerías públicas de la Universidad de Berkeley. No siempre encuentra la función mínima, pero se suele acercar lo suficiente, así que, siguiendo la más acendrada costumbre informática, "para qué vas a programar algo si alguien lo ha hecho antes...".

Y entonces, ¿hay algo que aportar en 2023 al algoritmo de Quine-McClusky, un algoritmo con casi 70 años de vida? Pues sí lo hay, de hecho se encuentran (con alguna dificultad) en la Red algunos artículos que explican determinadas estrategias que permiten bajar el consumo de CPU o memoria en ciertos pasos del algoritmo, aunque ninguna de ellas es *mainstream*, es decir, todos los artículos que explican el método, videos donde se desarrolla, etc. que he encontrado explican el método en su forma original.

Llevo algunos años trabajando en la optimización de programas fuente y, como parte fundamental de dicha optimización, llevo esos mismos años estudiando, implementando y en lo posible mejorando este famoso algoritmo, y he llegado a ciertas interesantes conclusiones que he decidido compartir con los lectores de El Cedazo y con todos aquellos que estén interesados en el tema.

No creo que vaya yo a descubrir ninguna primicia mundial, pero voy avisando desde ya de que, tal como se explica el algoritmo, y contra lo que se preconiza normalmente, **hay una probabilidad cercana al 50% de que la función obtenida por él no sea la mínima absoluta**, que es lo que se supone que debe entregar el algoritmo, en cuanto la forma canónica del problema tenga un cierto tamaño.

En las páginas siguientes voy a definir con toda la precisión que pueda el algoritmo de Quine-McClusky tal como lo tengo implementado en mis programas. Algunos pasos son los mismos que los del algoritmo original; otros son nuevos, no existen en el original; y otros, por fin, son diferentes a lo explicitado en la literatura. Todo lo iré explicando y defendiendo en su lugar correspondiente.

Una última reflexión: yo soy un informático actualmente jubilado, por lo que el documento está escrito como yo siempre he escrito documentos para otros informáticos (que solían entenderme) durante mi vida profesional: describiendo los procedimientos y pasos del algoritmo de forma procedimental. No vais casi a encontrar en este documento fórmulas matemáticas o lógicas; lo siento, pero no me siento cómodo con esa forma de definir los procesos informáticos y yo no la voy a usar aquí. Y cuando tenga que proponer y demostrar un teorema, lo haré lo mejor que pueda y sepa, intentando que se entienda bien el razonamiento, pero seguro que de una forma completamente alejada de lo que haría un colega con mejor formación que yo. Qué se le va a hacer, habrá que conformarse.

Por fin, una consecuencia directa de ser un viejo informático cascarrabias es que todo, todo lo que aquí escribo lo he probado. He programado cada rutina, cada proceso y cada instrucción que cito, y los resultados que expongo a lo largo de este documento son resultados reales de las diversas ejecuciones, millones de ellas, que he realizado. Y aseguro, además, que los programas que he usado no tienen errores de programación. Para ser exacto, tenían cientos de ellos, pero los he depurado todos hasta estar completamente seguro de que todos ellos funcionan correctamente. Si tuviera la más mínima duda al respecto no publicaría nada; uno no puede evitar ser como es.

A continuación citaré la nomenclatura que usaré a lo largo de este documento, y explicaré también cómo se confecciona la forma canónica, que es la única entrada para el algoritmo, a partir de una función lógica dada.

Nomenclatura utilizada

En Lógica normalmente se usan las letras *p*, *q*, *r*... para denominar las variables lógicas individuales, con una comilla final (a veces una raya sobre la letra) para expresar que son la negación de la variable: *p*', *q*', *r*', etc. Pero es ésta una nomenclatura poco útil a la hora de escribir programas que traten un conjunto de hasta 10, 12 o más variables distintas. Yo uso una nomenclatura más adecuada para mis propósitos: C1, C2, C3... etc. son las condiciones del programa, es decir, las variables afirmativas, mientras que N1, N2, N3, etc. son las negativas, las complementarias: N1 es la negación de C1, y por consiguiente C1, la negación de N1, etc. Para los operadores lógicos, usaré '+' para representar la suma lógica (OR) y '*' para el producto lógico (AND).

Así, las expresiones que aparecerán en este documento serán del tipo siguiente:
C2*N4+(N3+N4)*N5+C4*C5.

Siendo las condiciones, por ejemplo, C2: "*Año = 2022*"; C3: "*Mes > 8*"; C4: "*Código = 1*"; y por fin C5: "*Cancelado = NO*", la expresión anterior daría origen a:

"*Año = 2022*" AND "*Código NOT = 1*" OR ("*Mes NOT > 8*" OR "*Código NOT = 1*") AND "*Cancelado NOT = NO*" OR "*Código = 1*" AND "*Cancelado = NO*".

La forma canónica

El algoritmo de Quine-McClusky toma como entrada la forma canónica de la expresión lógica. La forma canónica es en realidad la tabla de verdad de la expresión. Para obtenerla, para cada posible valor de cada variable individual (0: Falso; 1: Cierto) se obtiene el valor de verdad de la expresión, que consta igualmente de ceros y unos. Mejor vemos un ejemplo de obtención de la tabla de verdad, con la misma expresión antes citada: **C2*N4+(N3+N4)*N5+C4*C5**.

Recuerdo brevemente que C1*C2 sólo es 1, Verdadero, si tanto C1 como C2 son verdaderos (sería C1 AND C2), y que C1+C2 sólo es 0, Falso, si tanto C1 como C2 son falsos (sería C1 OR C2). Y también que N1 tiene siempre el valor contrario al de C1: 0 si C1 es 1 y 1 si C1 es 0.

C2	C3	C4	C5	C2*N4	N3+N4	N5* (N3+N4)	C4*C5	C2*N4+ N5*(N3+N4)	C2*N4+N5* (N3+N4)+C4*C5
0	0	0	0	0	1	1	0	1	1
0	0	0	1	0	1	0	0	0	0
0	0	1	0	0	1	1	0	1	1
0	0	1	1	0	1	0	1	0	1
0	1	0	0	0	1	1	0	1	1
0	1	0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	1	0	1
1	0	0	0	1	1	1	0	1	1
1	0	0	1	1	1	0	0	1	1
1	0	1	0	0	1	1	0	1	1
1	0	1	1	0	1	0	1	0	1
1	1	0	0	1	1	1	0	1	1
1	1	0	1	1	1	0	0	1	1
1	1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	1	0	1

La forma canónica de la expresión C2*N4+(N3+N4)*N5+C4*C5 es, pues**,** la siguiente: **1011100111111101**, que es precisamente su tabla de verdad.

Obviamente, toda forma canónica válida tiene un tamaño de 2^n bits, siendo *n* el número de variables distintas que la componen.

Como la entrada al algoritmo es exclusivamente dicha forma canónica, se deben asignar nombres de variables para la emisión de la fórmula final: en este documento estas variables serán C1, C2, etc. o N1, N2, etc. si están complementadas, en forma negativa, hasta cumplimentar las *n* variables que contenga la forma canónica dada.

No creo que necesite explicar mucho más este proceso. Únicamente citaré unas cifras:

- La forma canónica de una expresión con *n* variables individuales tendrá 2^n posiciones, filas en la tabla anterior, con ceros o unos. La expresión de ejemplo tiene 4 variables individuales, C2...C5, por lo que su forma canónica tiene 16 posiciones, 2^4=16; para 5 variables son 32 posiciones (bits); para 10 variables son 1.024 posiciones, etc, es decir, su valor crece exponencialmente.

- En cuanto al número de posibles expresiones diferentes, es decir, formas canónicas distintas que se pueden construir con *n* variables, su número crece de forma doblemente exponencial: 2 elevado a 2^n. Para 3 variables son 256 posibles expresiones (2^8); para 4 variables son 65.536 (2^{16}). De ahí en adelante el número es ya imposible de tratar: para 5 variables (2^{32}) son ya más de 4.200 millones de expresiones diferentes; para 6 (2^{64}) son unos $1{,}8\cdot10^{19}$, o sea, 1.800 trillones, etc.

Es fácil demostrar que a partir de una cierta forma canónica puede obtenerse una expresión lógica equivalente: cada valor 1 de la forma canónica genera un término, un producto de las variables que dan origen a dicho uno, bien sea en forma afirmativa o negativa. A estos efectos, si la variable individual *Cx* tuviera un 0 en la forma canónica significa que dicha variable *Cx* está en su forma negativa, es decir, *Nx* en nuestra nomenclatura, y si *Cx* tuviera un 1, entonces está en forma afirmativa, *Cx* en nuestra nomenclatura.

Así, el primer 1 de la forma canónica del ejemplo (en la posición 1) da origen al término N2*N3*N4*N5, dado que los valores de las variables individuales que dan origen a dicho uno son todos ceros. Volved a consultar la tabla anterior para comprobarlo, si tenéis dudas. El segundo 1, que está en la posición 3, corresponde al término N2*N3*C4*N5, y así con los doce unos de la forma canónica hasta llegar al último uno, situado en la posición 16, que da origen el término C2*C3*C4*C5.

La suma lógica de estos doce productos es la **Forma Normal Disyuntiva** de la expresión. Esta Suma de Productos tiene 12 términos de 4 variables individuales cada uno, en sus formas afirmativa o negativa, por lo que la expresión consta de 48 variables individuales. Esta fórmula gigantesca es la que el algoritmo de Quine-McClusky sabe reducir a una que, teniendo la misma tabla de verdad, sea mínima, es decir, que contenga el mínimo número posible de variables individuales.

Cada variable individual, recordemos, representa una comparación que el programa tiene que realizar para determinar si el flujo del programa toma una u otra dirección en función de los valores concretos; cuantas menos comparaciones sean necesarias para determinar dicho flujo, más eficiente será el programa. Y en la industria de la electrónica de circuitos cada variable representa una puerta lógica que debe añadirse al circuito; cuantas menos variables, menos puertas lógicas, y por lo tanto más sencillo resulta el circuito.

Para terminar, recordemos que, como todo en álgebra de Boole es dual, también existe una **Forma Normal Conjuntiva**, es decir, un producto de sumas lógicas que también expresa la función original.

En el siguiente capítulo describiré con precisión el algoritmo de Quine-McClusky tal como lo tengo implementado, usando ciertas expresiones lógicas como ejemplos de cada paso. Sin embargo, por razones que se verán más adelante, no definiré en este capítulo el penúltimo paso del algoritmo, el que de verdad consume recursos informáticos de forma inmisericorde y que, por cierto, rara vez se explica convenientemente en los tutoriales que se encuentran en la Red.

En el tercer capítulo del documento explicaré un método alternativo para usar el algoritmo de Quine-McClusky que permite mejorar la obtención de la expresión mínima equivalente a la expresión dada, mejor dicho, que permite encontrar *siempre* dicha expresión mínima. Como antes cité, tal y como se define en la literatura hay una probabilidad elevada, cercana en muchos casos al 50%, de que el algoritmo no encuentre la función mínima absoluta.

Y, por fin, el último capítulo lo dedicaré a debatir sobre el penúltimo paso del algoritmo, el que es realmente costoso en uso de recursos informáticos, explicando los métodos usualmente citados para resolverlo en la literatura, así como diversas estrategias alternativas que he desarrollado para reducir el tiempo y la memoria necesaria para ejecutarlo.

Espero que todo este caudal de información que voy a exponer en este documento sea de utilidad para alguno de vosotros, los amables lectores de este documento.

Sois libres de utilizarlo como os plazca.

A ser posible, si no os importa, citando la fuente. Para alimentar mi ego.

Definición detallada del algoritmo de Quine-McClusky

El algoritmo de Quine-McClusky (QM como abreviatura) consta de una serie de pasos secuenciales, aunque cada uno de ellos tiene a su vez subpasos, iteraciones, selecciones y todo tipo de instrucciones más o menos complejas. Ésta es su definición detallada.

Paso 1. Detección y eliminación de variables "implicantes"

Este paso es nuevo, no existe, que yo sepa, en la definición standard del algoritmo. Yo, al menos, no he encontrado nada parecido.

Se trata de buscar si una o varias condiciones individuales de las contenidas en la fórmula original implican a la expresión completa, sea en su forma afirmativa, sea negada. Se buscan, en concreto, variables *Cx* o *Cy* que cumplan alguna de estas dos expresiones: *Cx*➔*Forig*, o bien, en su caso, *Ny*➔*Forig'*, siendo *Forig* la fórmula original, que de momento desconocemos, dado que sólo conocemos su forma canónica, y *Forig'* la negación de dicha fórmula original.

Veamos qué significa que exista una variable de este estilo en la fórmula original.

Cx➔*Forig*, es decir, *Cx* implica a *Forig* ocurre cuando la fórmula original tiene la forma *Forig=Cx+E,* es decir, cuando la variable *Cx* está sumando (OR) a otra cierta expresión *E* que da como resultado *Forig*; en cuanto a *Ny*➔*Forig'*, ocurre cuando la fórmula original tiene la forma *Forig=Cy*E*, o sea, la variable *Cy* está multiplicando (AND) a otra cierta expresión *E* que da como resultado *Forig*.

Si se localizara alguna de estas variables, entonces **dicha variable puede eliminarse completamente de la forma canónica**, reduciendo a la mitad su tamaño y, por consiguiente, reduciendo de forma dramática el uso de recursos necesarios para la ejecución del algoritmo.

Obviamente, no siempre este paso encuentra variables "implicantes" que eliminar, pero el coste de ejecución del paso es ridículo comparado con la posible ganancia conseguida en caso de encontrar una o varias variables que puedan eliminarse.

Este Paso 1 funciona de la forma siguiente:

1) Determinar cuántas variables individuales contiene la forma canónica en base al número de bits que la componen, teniendo en cuenta que dicho número de bits es siempre 2^n. Conocido dicho número *n*, componer la tabla de verdad de las *n* variables implicadas, tanto en su forma afirmativa como negada, y habilitar una tabla de variables implicantes en la que se anotarán las existentes con su operador relacionado.

2) Revisar la forma canónica original con las tablas de verdad de las *n* variables, tanto en su forma afirmativa como en su forma negada, comprobando si se cumple alguna de las dos condiciones siguientes, o ambas simultáneamente:
   a) Que todos los unos de la tabla de verdad de la variable tratada sean también unos en la forma canónica original.
   b) Que todos los ceros de la tabla de verdad de la variable tratada sean también ceros en la forma canónica original.

3) Si ninguna de las variables originales cumple alguna de las dos condiciones a) ó b), entonces terminar el paso y continuar el algoritmo por el Paso 2 con la forma canónica resultante, la original si no se ha encontrado ninguna variable implicante o la resultante tras la ejecución de este Paso 1.

4) En el caso de que para una cierta variable individual *Cx* se cumpla alguna de las dos condiciones a) ó b) citadas, entonces:
   4.1) Si se cumple la condición a) anterior, entonces llevar a la tabla de variables implicantes la propia variable *Cx* con el operador '+'.
   4.2) Si se cumple la condición b) anterior, entonces llevar a la tabla de variables implicantes la variable *Cx,* pero ahora con el operador '*'.
   4.3) Téngase en cuenta que la variable que cumpla alguna de las dos condiciones puede ser negativa, *Nx*. El procedimiento es idéntico en este caso, se lleva a la tabla de variables implicantes a esa *Nx* con el operador ('+', '*') que corresponda.
   4.4) Se eliminan de la forma canónica original todos aquellos unos o ceros que coinciden con los de la variable Cx, según se haya cumplido respectivamente la condición a) o b) anteriores. La forma canónica se reduce, por tanto, a la mitad, dado que todas las tablas de verdad de las variables individuales tienen el mismo número de ceros que de unos.
   4.5) Repetir todo el proceso desde el número 1, para intentar localizar nuevas variables implicantes en la forma canónica resultante, hasta que no se encuentren más o bien que toda la expresión haya sido resuelta.

5) En el caso particular de que en la comprobación de los ceros y los unos de una cierta variable individual *Cx* se cumplieran simultáneamente ambas condiciones a) y b), esto quiere decir que la tabla de verdad de la expresión original, o la resultante si ya se hubiera reducido mediante la ejecución del paso 4 anterior, es exactamente la misma que la de la variable individual *Cx*. En este caso la fórmula buscada es simplemente *Cx*; hay que pasar dicha variable *Cx* a la tabla de variables implicantes, pero sin signo, y parar inmediatamente el proceso: aunque resten variables sin eliminar en la forma canónica restante, son irrelevantes y pueden ignorarse.

6) Con la forma canónica resultante se ejecutan normalmente los pasos que restan del algoritmo QM, el cual obtiene finalmente una determinada fórmula. En el caso de que en este Paso 1 haya encontrado y extraído una o más variables implicantes, entonces deben ser incluidas en la fórmula final, extrayéndolas de la tabla de variables implicantes en el orden inverso al que fueron introducidas, es decir, dicha tabla es en realidad una pila (LIFO) en la que el último elemento que se introdujo será el primero en salir (Last Input, First Output). Las variables se incluyen sumando o multiplicando a la totalidad de la fórmula obtenida, según sea el operador que las acompaña.

Veamos un ejemplo de este Paso 1. Sea la forma canónica siguiente: **0000000010101110**. Son 16 bits, y por lo tanto tendremos en la expresión resultante 4 variables individuales, C1, C2, C3 y C4, puesto que $16=2^4$. En primer lugar se compone la tabla de verdad de las cuatro variables implicadas, tanto en su forma afirmativa como negativa. En este ejemplo de cuatro variables serían:

C1: 0000000011111111
N1: 1111111100000000
C2: 0000111100001111
N2: 1111000011110000
C3: 0011001100110011
N3: 1100110011001100
C4: 0101010101010101
N4: 1010101010101010

En la comparación de las diferentes tablas de verdad de las variables individuales, afirmativas o negativas, con la forma canónica, al compararla con la tabla de verdad de la variable C1 vemos que en todas las posiciones en las que hay un cero en dicha tabla de verdad también hay un cero en la forma canónica de la expresión original:

Forma canónica: **00000000**10101110
Tabla de verdad de C1: **00000000**11111111

Esto quiere decir que la variable C1 es implicante de la función original. Se lleva a la tabla de variables implicantes con el operador "*", dado que lo que coinciden son los ceros, y se eliminan de la forma canónica todos los bits correspondientes a dichos ceros de C1,que en este caso son los ocho primeros. El resultado de todo esto es:

Nueva forma canónica: 10101110

Var. implicante	Operador
C1	*

Como la nueva forma canónica tiene ahora 8 bits, sólo tiene 3 variables, luego también se deben eliminan las dos variables C1 y N1 ya procesadas de las tablas de verdad a comprobar, dado que esa variable ha sido incorporada a la tabla de variables implicantes y extraída de la función. Las tablas de verdad de las variables restantes, una vez eliminadas las variables C1 y N1, quedan así:

C2: 00001111
N2: 11110000
C3: 00110011
N3: 11001100
C4: 01010101
N4: 10101010

El paso se repite nuevamente hasta que no se encuentren más variables implicantes. Vemos ahora en esta segunda iteración que, al comparar la nueva forma canónica con las tablas de verdad, todos los unos de la variable N4 coinciden con unos en la forma canónica actual:

Forma canónica actual: **1**0**1**0**1**1**1**0
Tabla de verdad de N4: **1**0**1**0**1**0**1**0

Luego N4 es implicante también, esta vez sumando, dado que lo que coinciden ahora son los unos. Así, tras extraer la variable N4 (y también la C4, claro está) de la forma canónica y de la tabla de verdad de las variables restantes, y añadirla a la tabla de variables implicantes, queda:

Nueva forma canónica: 0010

Var. implicante	Operador
C1	*
N4	+

Y las tablas de verdad resultantes son:

C2: 0011
N2: 1100
C3: 0101
N3: 1010

Una nueva iteración del paso anterior encuentra ahora que todos los ceros de C2 coinciden con ceros en la forma canónica, luego C2 es implicante, multiplicando:

Forma canónica actual: **00**10
Tabla de verdad de C2: **00**11

Aplicando de nuevo los cambios preconizados, queda:

Nueva forma canónica: 10

Var. implicante	Operador
C1	*
N4	+
C2	*

Tablas de verdad resultantes:
C3: 01
N3: 10

La nueva forma canónica coincide completamente con N3, tanto los ceros como los unos, por lo que esta variable se debe incorporar también, pero sin signo. En este caso N3 es la última variable a tratar; toda la fórmula ha sido resuelta sin llegar siquiera a necesitar que se ejecute el resto del algoritmo.

La tabla de variables implicantes ha quedado definitivamente así:

Var. implicante	Operador
C1	*
N4	+
C2	*
N3	

Ahora, una vez finalizado este Paso 1, se pasaría a ejecutar el resto del algoritmo QM. En este ejemplo concreto, sin embargo, no hace falta, dado que toda la fórmula ha sido determinada en este paso y es obviamente mínima.

Por fin, a la hora de componer la expresión final, en el Paso 8, a la fórmula devuelta por el algoritmo **se añaden las variables implicantes, con su operador, en orden inverso al de introducción en la tabla**. En el ejemplo que hemos seguido primero se extrae N3, sin operador; posteriormente, C2 con el operador '*', lo que da N3*C2; a continuación se extrae N4 con el operador '+', lo que deja N3*C2+N4; y finalmente se extrae C1 con su operador '*', resultando así la fórmula definitiva: C1*(N3*C2+N4).

Si una vez extraídas todas las variables posibles la forma canónica resultante sí que tiene contenido, lo habitual, se ejecuta el resto de pasos del algoritmo QM. Tras su ejecución completa, a la expresión resultado del algoritmo se le deben añadir las variables implicantes de la forma antes descrita. Supongamos que la tabla de variables implicantes final ha quedado así:

Var. implicante	Operador
C7	*
N3	+
C4	+

Y supongamos también que el algoritmo ha devuelto una expresión *Eqm,* en la que, obviamente, no pueden aparecer C7, N3 ni C4, ni tampoco sus complementarias N7, C3 y N4. En ese caso la expresión final quedará:

(*Eqm*+C4+N3)*C7

Hasta aquí la descripción del **Paso 1, Detección y eliminación de Variables Implicantes**.

Este proceso de eliminación de variables implicantes, como ya he indicado, no lo he encontrado en ninguna publicación de las que he leído, y han sido decenas. Ignoro si soy el primero que lo define o si está ya descrito en algún lugar al que yo no haya accedido. En cualquier caso, ahí queda su definición por si algún colega desea implementarlo.

El coste de ejecución de este paso es mínimo comparado con la posible reducción a la mitad (o la cuarta parte, etc.) de la forma canónica a tratar, reduciendo en consonancia el tiempo necesario para tratarla a la raíz cuadrada del necesario, por lo que claramente compensa ejecutarlo aun cuando no encuentre ninguna variable implicante.

Con formas canónicas de 8 bits y tres variables individuales se encuentra al menos una variable implicante en 134 de las 256 formas canónicas posibles de ese tamaño; en formas canónicas de 16 bits y cuatro variables individuales se encuentran variables implicantes en 5.638 de las 65.536 formas canónicas posibles. Con tamaños mayores de forma canónica la proporción de éxito baja pero, dado su escaso coste, continúa siendo rentable buscar y eliminar las posibles variables implicantes en términos de consumo de recursos.

A continuación describiré el resto de pasos del algoritmo de Quine-McClusky tal como se define en la literatura, explicando en cada paso, si procede, lo que he ido aprendiendo en mis investigaciones y pruebas.

Para describir los pasos restantes del algoritmo me apoyaré en principio en un ejemplo, uno cuya forma canónica es la siguiente:

10111001111111011111111111111111

Son 32 bits, cuatro ceros y 28 unos, y por tanto contiene cinco variables individuales, C1...C5, dado que $2^5 = 32$.

En el primer paso del algoritmo, la eliminación de variables implicantes que ya hemos visto, comprobamos que todos los unos de la tabla de verdad de la variable C1 coinciden con unos en la forma canónica dada:

Forma canónica: 1011100111111101**1111111111111111**
Tabla de verdad de C1: 0000000000000000**1111111111111111**

Por lo tanto C1 es implicante de la función completa. Se extrae C1 de la forma canónica (en este caso son sus últimos 16 unos), y se lleva, sumando, a la tabla de Variables Implicantes:

Var. implicante	Operador
C1	+

En la forma canónica de 16 bits y cuatro variables resultante no es ya posible encontrar más variables implicantes, por lo que dicha forma canónica resultante, que por consiguiente es la que se toma como entrada en los pasos restantes del algoritmo, es la siguiente:

1011100111111101

La forma canónica a tratar tiene ahora 16 bits, 12 de los cuales son unos y los 4 restantes, ceros. También son cuatro las variables individuales involucradas, de C2 a C5. Recordemos que la variable C1 ha resultado ser implicante y que por ello ha sido extraída de la original; no se puede reutilizar su nombre a lo largo del algoritmo, pues en caso contrario la expresión final no será correcta cuando se añada como último paso dicha condición C1 sumando a la expresión finalmente obtenida.

Paso 2- Carga inicial de los *minterms*

Antes de describir este paso necesito aclarar una cierta terminología que se usa continuamente en el algoritmo: *minterm* e *implicante primo*.

Un *minterm* (término mínimo) es cada una de las combinaciones que dan origen a un 1 en la forma canónica. Así, el tercer uno de la forma canónica anterior, que se encuentra en la cuarta posición de la forma canónica, da origen biunívocamente al *minterm* 0011 (N2*N3*C4*C5). En el algoritmo sólo se usa el índice 0011, su conversión al término N2*N3*C4*C5 está implícita. Posteriormente se verá en detalle esta conversión. El nombre "*minterm*" viene porque basta con que uno solo de estos términos se cumpla para que la función completa se cumpla, consecuencia obvia de que todos ellos están sumados (OR) entre sí para conformar la función global.

Existen también los "*maxterms*", en caso de que la función se definiera en Forma Normal Conjuntiva, un Producto de Sumas: cada término, una suma de variables individuales, es un *maxterm* porque, al estar todos ellos multiplicados (AND) entre sí, se precisa que todos y cada uno sean ciertos para que sea cierta la función global.

Un *implicante primo* (IP a partir de ahora) es el resultado de fusionar varios *minterms*, y se llama de esta forma porque cada uno de ellos implica a la función original, es decir, cada uno de los IPs cumple que *IP➔Exp.Original*. Veremos cómo se forman un poco más adelante.

Definición del Paso 2: Para cada *minterm* de la forma canónica dada, es decir, **para cada uno que allí aparece**, se carga un elemento en una tabla base de implicantes primos, calculando adicionalmente el número de unos que tiene el *minterm*.

En nuestro ejemplo la forma canónica tiene doce unos, por lo que doce son los elementos que se cargarán inicialmente en la tabla. Son estos:

Num.unos	Implicante primo
0	0000
1	0010
1	0100
1	1000
2	0011
2	1001
2	1010
2	1100
3	0111
3	1011
3	1101
4	1111

Como se puede ver, además del valor del *minterm* se ha cargado también el número de unos que tiene dicho implicante primo. "0000" no tiene ningún uno, todos son ceros, mientras "0111" tiene tres unos, etc. Esto será importante para el paso siguiente, como veremos en un momento.

La tabla se clasifica por el número de unos y el valor del implicante primo, quedando como se puede ver más arriba. Sí, en este punto un *minterm* y un *implicante primo* es lo mismo; a partir de ahora ya serán diferentes.

Paso 3 – Cálculo de los implicantes primos

Para cada uno de los implicantes primos de la tabla base, compararlo con todos y cada uno de los IPs de la propia tabla que tengan un número de unos exactamente mayor en uno al del IP comparado.

Así, el IP "0000", con cero unos, se comparará con todos los IPs de la tabla que tengan un solo uno, es decir, con "0010", "0100" y "1000", pero no con el resto. Y así con todos y cada uno de los Implicantes Primos de la tabla.

En el proceso se generan nuevos elementos en otra tabla similar, llamémosla "tabla nueva" para distinguirla de la tabla base que se procesa, en la que **se genera un nuevo elemento cuando los dos elementos comparados sean exactamente iguales excepto en una única posición**.

Si eso ocurre **se genera en la tabla nueva un elemento igual a cualquiera de ellos, pero con un guión (–) en la única posición en la que ambos difieren**. Así, de la comparación entre "0000" y "0010" se obtiene el nuevo elemento "00–0", donde se ha sustituido el tercer carácter, que es el que difiere, por un guión. En dicho elemento de la tabla nueva se añade también el número de unos que tiene dicho nuevo elemento, calculado con el mismo criterio que en la carga inicial. Si hubiera más de una diferencia entre los dos elementos comparados no se genera ningún elemento en la tabla nueva.

El fundamento de este paso es que comparar dos elementos de la tabla equivale a tratar dos términos, dos productos de la Forma Normal Disyuntiva (FND) de la expresión dada. Cuando se encuentra que sólo se diferencian en un bit, en un elemento es un uno y un cero en el otro, esto implica que se han encontrado dos términos que pueden ser fusionados aplicando la propiedad distributiva, es decir, sacando factor común entre ambos y eliminando la variable cuyos valores difieren.

Sean, por ejemplo, los elementos 0101 y 1101 los que estamos comparando. Sólo se diferencian en un único bit, el primero de ellos, y ambos darán origen, según se explicó antes, al IP –101. Estos dos elementos representan respectivamente a N2*C3*N4*C5 y C2*C3*N4*C5. Como ambos términos están sumados en dicha FND, se puede aplicar la reducción siguiente, sacando como factor común el término común C3*N4*C5 de la forma:

N2*C3*N4*C5 + C2*C3*N4*C5 = (N2+C2)*C3*N4*C5 = (1)*C3*N4*C5 = C3*N4*C5;

Esto es exactamente lo que representa el implicante primo –101, que es el resultado de la comparación: que pueden fundirse estos dos términos en uno más sencillo que los engloba a ambos.

Volvamos de nuevo a nuestro ejemplo.

La primera pasada de comparaciones y sus resultados son los siguientes (recordemos que cada elemento sólo se compara con todos los elementos que tienen exactamente un uno más que el comparado):

(0) 0000 – (1) 0010 = 00–0
(0) 0000 – (1) 0100 = 0–00
(0) 0000 – (1) 1000 = –000
(1) 0010 – (2) 0011 = 001–
(1) 0010 – (2) 1001 = **NO** (hay más de una diferencia, no se genera elemento en la tabla nueva)
(1) 0010 – (2) 1010 = –010
(1) 0010 – (2) 1100 = **NO**
(1) 0100 – (2) 0011 = **NO**
(1) 0100 – (2) 1001 = **NO**
(1) 0100 – (2) 1010 = **NO**
(1) 0100 – (2) 1100 = –100
(1) 1000 – (2) 0011 = **NO**
(1) 1000 – (2) 1001 = 100–
(1) 1000 – (2) 1010 = 10–0
(1) 1000 – (2) 1100 = 1–00
(2) 0011 – (3) 0111 = 0–11
(2) 0011 – (3) 1011 = –011
(2) 0011 – (3) 1101 = **NO**
(2) 1001 – (3) 0111 = **NO**
(2) 1001 – (3) 1011 = 10–1
(2) 1001 – (3) 1101 = 1–01
(2) 1010 – (3) 0111 = **NO**
(2) 1010 – (3) 1011 = 101–
(2) 1010 – (3) 1101 = **NO**
(2) 1100 – (3) 0111 = **NO**
(2) 1100 – (3) 1011 = **NO**
(2) 1100 – (3) 1101 = 110–
(3) 0111 – (4) 1111 = –111
(3) 1011 – (4) 1111 = 1–11
(3) 1101 – (4) 1111 = 11–1

La nueva tabla ha quedado, por lo tanto y una vez clasificada, de la forma siguiente:

Num.unos	Implicante primo
0	–000
0	0–00
0	00–0
1	–010
1	–100
1	001–
1	1–00
1	10–0
1	100–
2	–011
2	0–11
2	1–01
2	10–1
2	101–
2	110–
3	–111
3	1–11
3	11–1

En el caso de que tras todas las comparaciones hubiera quedado algún elemento de la tabla base que no haya podido ser fusionado con ningún otro, entonces también se copian todos ellos a la nueva tabla. En este ejemplo concreto todos los elementos se han fusionado con otros una o varias veces, por lo que no hay que añadir ninguno adicional a la tabla nueva.

Ya se puede ver en el ejemplo que eran doce los IPs originales, pero ha habido que hacer 30 comparaciones en esta primera iteración del paso y se han generado 18 elementos, en los que se sigue añadiendo el número de unos que contiene cada elemento.

La nueva tabla se ordena de nuevo por el número de unos y el valor del IP, se eliminan los elementos duplicados que pudiera haber (en esta primera iteración del paso no hay ninguno) y se copia la nueva tabla resultante a la tabla base, para repetir el proceso de este Paso 3 tantas veces como sea necesario hasta que no se haya conseguido reducir ni un solo IP más.

La nueva tanda de comparaciones es tediosa de escribir (esta vez son 72 comparaciones), así que sólo escribiré la nueva tabla de los IPs resultantes. Habrá que creerme, o mejor dicho, al programa que lo hace. Nuevamente, en esta iteración todos los elementos de la tabla han sido fusionados con algún otro, por lo que no se debe copiar ningún elemento adicional no fusionado.

El resultado de esta segunda iteración del Paso 3 es que en la nueva tabla quedan 14 IPs:

Num.unos	Implicante primo
0	–0–0
0	—00
0	—00
0	–0–0
1	–01–
1	–01–
1	1–0–
1	10—
1	10—
1	1–0–
2	—11
2	—11
2	1—1
2	1—1

Pero en realidad son sólo 7 IPs diferentes, que en este caso se repiten dos veces cada uno. Una vez ordenada la tabla y eliminados los duplicados, la tabla queda así:

Num.unos	Implicante primo
0	—00
0	–0–0
1	–01–
1	1–0–
1	10—
2	—11
2	1—1

La tercera iteración del Paso 3 realiza 12 comparaciones pero, como puede verse fácilmente, ninguna encuentra que los dos elementos comparados sólo difieran en una única posición, por lo que no hay más fusiones y, por lo tanto, ésta es la tabla de implicantes primos definitiva.

Este paso es terrible en uso de recursos en cuanto el número de variables individuales crece. Entre las muchas pruebas que he hecho, usé una expresión de 10 variables individuales cuya forma canónica tenía 1.006 unos de los 1.024 (2^{10}) posibles.

Según la información existente en la red, el número de implicantes primos a calcular para una función de *n* variables puede llegar a ser de 3^n dividido por la raíz cuadrada de *n*, es decir, según esta fórmula en esta función de 10 variables podemos esperar que el número de implicantes primos llegue a ser de hasta 18.673. Veamos ahora la realidad de esta función:

Como el valor 1 se da en 1.006 ocasiones en la forma canónica de la expresión, la expresión tiene 1.006 *minterms*, y por ello la carga inicial de implicantes primos cargó exactamente 1.006 implicantes primos. Para tratarla fueron necesarias diez iteraciones del bucle de este Paso 3, y la tabla de implicantes primos llegó a tener en la cuarta iteración más de 46.000 implicantes primos. En concreto, la dimensión de la tabla de implicantes primos en cada iteración fue de 1.006, 4.964, 21.854, 42.180, 46.496, 31.900, 13.884, 3.724, 560, 40 y, finalmente, la tabla de implicantes primos simplificados resultado final de este Paso 3 tenía solamente 8 implicantes primos. Pero para llegar a esos magros ocho implicantes primos finales se han tenido que realizar por el camino muchos millones de operaciones. El método de ordenación utilizado debe ser Quicksort; otro método daría resultados desastrosos. Y, por cierto, el número máximo de implicantes primos que se requieren en este caso para completar el Paso 3 en este caso, 46.496, es dos veces y media mayor que el supuesto límite superior teórico establecido de dicho número.

En mis búsquedas por la Red he encontrado algunas publicaciones donde se explican diversas estrategias para mejorar el rendimiento de este Paso 3 y que consuma menos recursos, alguna basada en comparar los elementos de la tabla sólo cuando se cumplan unas ciertas restricciones para no tratar elementos que de antemano se determine que no van a fusionarse, o bien, en otro caso, en aplicar matrices de tamaño 3^k para reducir el número de comparaciones, y con toda probabilidad existen más aproximaciones que yo no he encontrado para reducir el coste de este Paso 3. Sin embargo, aunque no he implementado ninguna de estas ideas en mis programas, mi instinto de viejo informático me hace pensar que el coste de determinar si se compara o no un elemento con otro debe ser casi el mismo que el de hacer físicamente la comparación. Lo cierto es que en la práctica totalidad de publicaciones sobre el algoritmo QM no se cita ninguna de estas mejoras.

Lo que sí se cita de forma casi universal es que cuando se fusionan dos IPs para dar origen a un nuevo IP se debe apuntar cuidadosamente qué *minterms* dan origen a este nuevo IP fusionado, para posibilitar o simplificar el Paso 4 siguiente. Por ejemplo, al fusionar el IP 0000 con el 0100 para originar el nuevo IP 0–00, se debe anotar asociado a este IP 0–00 de dónde viene: de los *minterms* 0000 y 0100. Lo normal es que se guarden estos números en notación decimal, 0 y 4, supongo que para hacerlo más sencillo a la vista que en binario. En iteraciones sucesivas, por ejemplo cuando se fusiona 0–00 con 1–00 para dar origen a —00 se debe anotar en dicho IP los cuatro *minterms* que cubre: 0000 y 0111, herencia de 0–00, y 1000 y 1100, herencia de 1–00. Según se van haciendo más y más fusiones, el tamaño de estas listas de *minterms* crece y crece... listas de un número variable de elementos que se vuelven muy costosas de mantener, tanto por su configuración variable como por el tamaño de cientos de miles o millones de elementos que puede alcanzar la suma de las diferentes listas.

Sin embargo, **esta costosa técnica de llevar las listas de *minterms* no sirve para nada**, como veremos en seguida, y no debe implementarse de ninguna forma en el algoritmo.

Paso 4 - Carga de la tabla de cobertura de *minterms*

En base a la lista de implicantes primos simplificados por el paso anterior se rellena una tabla bidimensional en la que, en columnas, tendremos una por cada *minterm* que tenga la expresión original, es decir, una por cada uno de la forma canónica de la expresión, doce en nuestro ejemplo, y en filas tendremos una por cada implicante primo simplificado resultante. En las celdas de la tabla se marca con una X cuando el implicante primo de la fila cubre al *minterm* de la columna. Para saber qué *minterms* cubre cada IP basta con desarrollar el propio IP, sustituyendo cada guión por sus dos posibles valores 0 y 1.

Así, un IP como el --00, que tiene dos guiones, cubre 4 *minterms* (2^2), que son: 0000, 0100, 1000 y 1100, resultado de sustituir ambos guiones por las 4 combinaciones posibles de 0 y 1. Con esta sencilla técnica, tal como avanzaba en la página anterior, se evitan las interminables listas de *minterms* cubiertos por cada implicante primo, pues ésta es su única utilidad: conocer qué *minterms* cubre cada IP para marcar en consecuencia las equis en la tabla de cobertura.

Es importante añadir a la tabla las etiquetas, tanto en filas como en columnas, dado que, como veremos, en los siguientes pasos se irán eliminando tanto filas como columnas de la tabla.

En el ejemplo que hemos seguido hasta ahora, con siete implicantes primos finales y doce *minterms*, y una vez ordenados los IPs, la estructura de la tabla de cobertura quedaría así:

IP	0000	0010	0011	0100	0111	1000	1001	1010	1011	1100	1101	1111
—00	X			X		X				X		
—11			X		X				X			X
–0–0	X	X				X		X				
–01–		X	X					X	X			
1—1							X		X		X	X
1–0–						X	X			X	X	
10—						X	X	X	X			

Los pasos siguientes intentarán cubrir todos los *minterms*, las columnas, utilizando para ello el menor número posible de implicantes primos, aunque para nuestro objetivo final de optimizar el código fuente en realidad lo que más interesa es usar el menor número posible de variables individuales (las condiciones del programa), es decir, el número de caracteres que no sean guiones de los IPs seleccionados. Recordemos que cada guión de un implicante primo representa una variable que ha sido eliminada, mientras que cada cero o uno de un implicante primo representa a una variable individual, negada o no, es decir, una condición a comprobar en la función lógica; cuantas menos haya de éstas, más rápida será la evaluación de la expresión total.

Es claro que en este caso el tamaño, fundamentalmente el número de IPs de la tabla de cobertura recién creada, importa. Cuando crece por encima de ciertos valores se necesita un enorme consumo de recursos para solventarlo. Es por eso que los dos siguientes pasos del algoritmo lo que intentan es reducir en lo posible el número de elementos de la tabla, tanto filas como columnas, para simplificar así la obtención de la solución final.

Paso 5 - Detección de Implicantes Primos Esenciales

Si nos fijamos en la tabla anterior nos damos cuenta de que hay algunos *minterms* que tienen una única X en su columna. Es el caso, por ejemplo, del *minterm* 0100: sólo el IP —00 lo cubre, así que obligatoriamente dicho IP tiene que estar en la lista final de IPs, pues en otro caso quedaría sin cubrir dicho *minterm* 0100. A estos IPs se los denomina *Implicantes Primos Esenciales*, y deben todos ellos formar parte obligatoriamente de la lista final de IPs buscada.

Una vez seleccionado un implicante primo esencial, se lleva a la tabla de IPs seleccionados, se elimina su fila de la tabla y también se eliminan todas las columnas en las que ese IP tenga una X, dado que los *minterms* correspondientes ya han sido cubiertos por ese IP seleccionado.

Bien: el IP —00 es el único que cubre el *minterm* 0100, luego es esencial y se selecciona.

Se lleva a la tabla de IPs seleccionados, se elimina la fila del propio implicante primo y también las columnas de los *minterms* que cubre, y las tablas quedan así:

IP	0010	0011	0111	1001	1010	1011	1101	1111
10—				X	X	X		
—11		X	X			X		X
–0–0	X				X			
–01–	X	X			X	X		
1—1				X		X	X	X
1–0–				X			X	

IPs Seleccionados
—00

Un pequeño truco aquí es desplazar el último IP de la tabla a la posición del IP eliminado en vez de mover todas las filas, ahorrando algo de tiempo de CPU. Por eso el IP 10— aparece ahora en la primera fila en vez de en la séptima original. Sigamos.

Nuevamente el IP —11 es ahora el único que cubre el *minterm* 0111, por lo tanto es también esencial y debe ser seleccionado. Se elimina dicho implicante primo y las columnas de los *minterms* que cubre, y por lo tanto quedan ahora en la tabla de cobertura cinco IPs:

IP	0010	1001	1010	1101
10—		X	X	
1–0–		X		X
–0–0	X		X	
–01–	X		X	
1—1		X		X

IPs Seleccionados
—00
—11

Ya no hay más implicantes primos esenciales, todos los *minterms* están cubiertos por al menos dos implicantes primos. Y todos los IPs esenciales localizados hasta ahora han sido trasladados a la tabla de IPs seleccionados y eliminados de la tabla de cobertura.

Por supuesto es posible, y ocurre con frecuencia que, dada una cierta forma canónica y llegados a este momento, no haya ningún IP esencial que pueda ser eliminado; sin embargo, y tal como comentaba en el Paso 1, el coste de realizar esta búsqueda de implicantes primos esenciales es muy pequeño comparado con la posible reducción de tamaño en la tabla de cobertura, que minorará significativamente el tiempo necesario para ejecutar el resto de pasos.

Paso 6 – Eliminación de Implicantes Primos redundantes

En este Paso se intenta eliminar de la tabla de cobertura aquellos IPs de los restantes que sean redundantes con otros implicantes primos de la propia tabla. Un IP IP_1 es redundante con otro IP_2 cuando dicho IP_2 cubre al menos todas las columnas (*minterms*) que cubre IP_1, pudiendo IP_2 cubrir, además algunos otros adicionales.

La justificación es sencilla: supongamos que tenemos la tabla de cobertura siguiente, donde IP_1 cubre dos columnas (*minterms*), la columna 2 y la 3, mientras que IP_2 cubre las columnas 2, 3 y 4.

IP	1	2	3	4
IP_1		X	X	
IP_2		X	X	X

Aquí todos los *minterms* que cubre IP_1, 2 y 3, los cubre también IP_2, que adicionalmente cubre un *minterm* más, el 4. Por lo tanto, es seguro que seleccionando IP_2 se cubren eficazmente todas las columnas que cubre IP_1. La consecuencia es que el implicante primo IP_1 es redundante con IP_2, que *lo domina*, según la terminología usada en la literatura, y por lo tanto se puede suprimir tranquilamente dicho implicante primo IP_1 de la tabla de cobertura.

Esta eliminación sólo es posible si en el Paso 5 se hubiera encontrado algún implicante primo esencial; si no hubiera ninguno entonces es seguro, debido al método de fusión de IPs seguido en el Paso 3, que no hay IPs redundantes y por tanto que ningún IP cubre a ningún otro.

Además, una consecuencia de esta posible eliminación de IPs redundantes es que ciertos IPs que antes no lo eran se conviertan en esenciales, denominados *implicantes primos esenciales secundarios*, pero que por muy secundarios que sean son igual de esenciales que los anteriores.

Esto implica que **la forma correcta de implementar los Paso 5 y 6 es en realidad juntos**, **en una iteración** que se repite hasta que no se hayan encontrado más IPs esenciales en la tabla de cobertura, de la forma:

Paso conjunto 5-6:

1) Ejecutar el proceso de búsqueda y eliminación de IPs esenciales (**Paso 5**).
2) Si no se ha encontrado ningún IP esencial, terminar el Paso conjunto 5-6.
3) Ejecutar el proceso de búsqueda y eliminación de IPs redundantes (**Paso 6**).
4) En el caso de que se haya eliminado al menos un IP redundante, volver al paso 1; en el caso de no haber eliminado ninguno, terminar el Paso conjunto 5-6.
5) Terminar también en el caso de que no queden ya columnas en la tabla de cobertura, por haber sido cubiertas todas ellas por los implicantes esenciales seleccionados.

Volvamos ahora a nuestro ejemplo, donde restaban cinco IPs y cuatro *minterms* en la tabla de cobertura, y donde habían sido seleccionados dos IPs esenciales hasta el momento. Ésta era la situación:

IP	0010	1001	1010	1101
10—		X	X	
1–0–		X		X
–0–0	X		X	
–01–	X		X	
1—1		X		X

IPs Seleccionados
—00
—11

Al buscar IPs redundantes vemos que el IP 1–0– cubre exactamente las mismas columnas que el IP 1—1, los *minterms* 1001 y 1101. Puede eliminarse uno cualquiera de ellos, dado que en este punto son idénticos, pues cubren las mismas columnas. Eliminemos el último, el 1—1.

Por otra parte, el IP –0–0 cubre a su vez las mismas columnas que el IP –01–, que es también eliminado.

Como consecuencia de ambas eliminaciones de IPs redundantes la tabla de cobertura de IPs queda ahora así:

IP	0010	1001	1010	1101
10—		X	X	
1–0–		X		X
–0–0	X		X	

Ahora los IPs 1–0– y –0–0 se han convertido en esenciales secundarios, pues son los únicos que cubren a los *minterms* 1101 y 0010, respectivamente.

Por tanto, se seleccionan ambos y se llevan a la tabla de IPs seleccionados, que queda así:

IPs Seleccionados
—00
—11
1–0–
–0–0

Y ahora, al eliminar estos dos últimos implicantes primos, todos los *minterms* han resultado cubiertos. Ya no quedan columnas en la tabla. En ella queda un solo IP, el 10—, que es irrelevante y se ignora. No es necesario proseguir con el resto del algoritmo, lo que sí sería necesario en el caso habitual de que quede en la tabla de cobertura un cierto número de *minterms* sin cubrir por un cierto conjunto de IPs restantes. Habrá entonces que seleccionar de alguna manera un conjunto mínimo de IPs que cubran todos los *minterms* restantes.

Qué hacer en el caso de que la tabla de cobertura sí que tenga contenido tras la ejecución de los Pasos 5 y 6, lo que es el caso general, es decir, lo que será el **Paso 7** del algoritmo de Quine-McClusky, **Búsqueda de la Solución Óptima**, será tratado por razones didácticas más adelante en este documento

Los Pasos 2 a 6 que he explicado hasta ahora son, con mínimos ajustes, los mismos pasos descritos en la literatura sobre el algoritmo QM, aunque numerados de 1 a 5, dado que el Paso 1 tal como yo lo he definido no existe o al menos no lo he encontrado en dicha literatura.

Ahora, antes de seguir el desarrollo del resto del algoritmo debo comentar algunas de las conclusiones y hechos que he encontrado en mis pruebas e investigaciones. Para ello necesito adelantar la definición del **Paso 8, Obtención de la Solución Final Óptima**. De hecho, en casos como el del ejemplo que hemos seguido hasta aquí no es necesario ejecutar el antes citado Paso 7, dado que todos los *minterms* han sido cubiertos con los IPs esenciales, por lo que el devenir natural del algoritmo pasaría en este caso concreto por la ejecución directa de dicho Paso 8, que paso a definir inmediatamente.

Paso 8 – Obtención de la expresión óptima final

En base a los implicantes primos finalmente seleccionados para formar parte de la solución final y contenidos en la "Tabla de IPs seleccionados", para obtener la expresión resultante como suma de productos basta con representar, multiplicando (AND) entre ellas, las variables referenciadas por cada implicante primo de la tabla final, en afirmativo si hay un 1, y en negativo si hay un cero, y obviando los guiones que no dan origen a ninguna variable, y luego sumar (OR) todos los términos resultantes.

Una vez obtenida de esta forma la expresión final, hay que añadir a la expresión todas las variables implicantes encontradas en el Paso 1 – Detección de Variables Implicantes. Estas variables implicantes se añaden a la expresión en el orden inverso al que fueron detectados, conectándolas con la expresión existente sumando/multiplicando a la totalidad de la expresión existente, según sea '+' o '*', respectivamente, el operador almacenado junto a la variable implicante en su tabla correspondiente.

En nuestro ejemplo, recordemos que la primera posición de los IPs corresponde a la variable C2, la segunda, a C3, la tercera a C4 y la cuarta y última a C5, ya que la variable C1 era implicante y había sido eliminada de la forma canónica en el Paso 1. Como los IPs finalmente seleccionados son cuatro, cuatro serán los productos obtenidos, y como todos ellos tienen dos bits válidos y dos guiones, todos los productos tendrán dos variables.

En el primer IP seleccionado, que es —00, la primera posición, correspondiente a C2, es un guión, luego se ignora, lo mismo que el segundo guión, correspondiente a C3. El tercer bit, el de C4, es un cero, luego la variable está negada, es N4, y lo mismo con el cuarto bit, el de C5, que al ser otro cero está negada también, luego es N5. El término generado por —00 es, por tanto, N4*N5. En definitiva, los términos resultantes son los siguientes:

IPs Seleccionados	Términos generados
—00	**N4*N5**
—11	**C4*C5**
1–0–	**C2*N4**
–0–0	**N3*N5**

Los cuatro términos son, pues: N4*N5, C4*C5, C2*N4 y N3*N5 y, como paso final, todos estos productos se suman entre sí para dar origen a la expresión definitiva:

N4*N5+C4*C5+C2*N4+N3*N5

Aquí se quedan la práctica totalidad de publicaciones sobre el algoritmo QM, indicando que la expresión así obtenida es la mínima posible. Sin embargo, es obvio que **una suma de productos puede reducirse de tamaño simplemente aplicando la propiedad distributiva**, es decir, sacando como factores comunes aquellas variables que estén en diferentes términos, cosa que es habitual que se dé en la expresión obtenida, que es una Suma de Productos.

Así, esta expresión de 8 variables individuales es susceptible de ser reducida un poco más sacando factor común, en este caso a N5 (también se podría hacer con N4, pero no con ambas a la vez). Reordenando, esta expresión mínima final es:

C2*N4+N5*(N3+N4)+C4*C5

Expresión final que consta de tan solo 7 variables individuales, y que, ahora sí, ya no es posible reducir más.

Recordemos, sin embargo, que aún faltan por incluir las variables implicantes detectadas y eliminadas en el Paso 1. Como paso final, éstas deben ser incluidas en la fórmula final en el orden inverso a su eliminación (LIFO), conectadas con la expresión final por el operador que acompaña a dichas variables implicantes en su tabla correspondiente.

En nuestro ejemplo había una sola variable implicante, C1, con el operador '+', variable que hay que sumar a la totalidad de la expresión encontrada hasta el momento, por lo que la expresión definitiva es: **C1+C2*N4+N5*(N3+N4)+C4*C5.**

Ésta sí que es, definitivamente, la expresión mínima absoluta correspondiente a la forma canónica de ejemplo, 10111001111111011111111111111111.

O al menos *eso es lo que dicta la teoría unánimemente aceptada*.

Pero no es así, porque, atención: **ésta no es la expresión mínima equivalente a la forma canónica dada**. Para obtener de una vez por todas la que sí que es la expresión mínima absoluta de la forma canónica dada en el ejemplo, 10111001111111011111111111111111, es preciso utilizar el mismo algoritmo QM, pero *de una forma ligeramente distinta*.

Proceso alternativo del algoritmo Quine-McClusky

El capítulo anterior terminaba con una afirmación contundente y seguramente sorprendente: "...**ésta no es la expresión mínima equivalente a la forma canónica dada**. Para obtener de una vez por todas la que sí que es la expresión mínima absoluta de la forma canónica dada en el ejemplo, 10111001111111011111111111111111, es preciso utilizar el mismo algoritmo QM, pero *de una forma ligeramente distinta*".

Esta afirmación es ciertamente contundente, pues implicaría de ser cierta que el algoritmo de Quine-McClusky tal como se presenta en la literatura no siempre da como resultado la expresión mínima equivalente. Y eso hay que probarlo.

Pues a eso vamos. Aunque ya voy adelantando que en mis investigaciones he encontrado que, dependiendo del tamaño de la forma canónica, *hay una probabilidad cercana al 50% de que el algoritmo no obtenga la función mínima buscada*. Es decir, que casi una de cada dos veces el algoritmo falla en obtener la expresión mínima equivalente a la dada.

Describamos en detalle, pues, por qué en ocasiones el algoritmo QM falla en encontrar la función mínima y cuál es ese Proceso alternativo que, ahora sí, encuentra siempre la función mínima equivalente a la forma canónica dada.

El algoritmo, como hemos visto hasta aquí, se basa en tratar la forma canónica de la expresión a minimizar: toma los unos de dicha forma canónica para generar los *minterms* y cargarlos en la tabla, luego fusionarlos para conseguir los implicantes primos, etc.

Preguntémonos ahora qué ocurriría si **en vez de tomar los unos de la forma canónica tomáramos los ceros**, y luego se siguieran los pasos del algoritmo exactamente igual que lo descrito hasta ahora.

El resultado de esa forma de proceder sería obviamente **la fórmula mínima que daría origen a la *expresión complementaria a la buscada***. Entonces, una vez obtenida esta expresión mínima complementaria, **para obtener la expresión buscada bastaría con negar la expresión complementaria así calculada**, aplicando las Leyes de De Morgan. Recuerdo que las Leyes de De Morgan preconizan que (a*b)'=a'+b', y que (a+b)'=a'*b'.

En fin, la expresión resultante sería un Producto de Sumas en vez de una Suma de Productos. Y sería igual de "mínima" que la obtenida por el método tradicional... o quizá no. Enseguida veremos si esto es así.

Ignoro las implicaciones que en el diseño de circuitos electrónicos tiene el que la expresión sea un producto de sumas en vez de una suma de productos, pero puedo asegurar que si lo que se desea es optimizar el código de una expresión lógica contenida en un programa fuente da exactamente igual una u otra estructura de la expresión: lo importante es reducir al máximo el número de condiciones de la fórmula para minimizar el número de comparaciones a realizar para determinar el flujo del programa. Si hay más ORs que ANDs o viceversa no importa lo más mínimo, tan sólo importa reducir el número de condiciones todo lo posible.

En fin, tratar los ceros en vez de los unos implica tratar no los *minterms* de la expresión, sino los *maxterms*. Más adelante hablaré sobre las referencias que he encontrado sobre el uso de *maxterms* en la literatura sobre el algoritmo de Quine-McClusky.

Llevando a la práctica esta orientación de tratar los ceros en vez de los unos de la forma canónica, vamos a ejecutar los diferentes pasos del algoritmo tal como se definieron en el capítulo anterior para el mismo ejemplo que hemos seguido hasta ahora. Será rápido.

El **PASO 1** se ejecuta exactamente igual que antes, puesto que si una variable resulta ser implicante, lo será independientemente de que el resultado del algoritmo sea una Suma de Productos o un Producto de Sumas. Por lo tanto, igual que antes, la variable C1, y sólo ella, es implicante, sumando, y se extrae de la forma canónica.

Tras esta extracción de C1, recordemos, la forma canónica resultante era la siguiente: **1011100111111101**, y las variables individuales involucradas son C2, C3, C4 y C5, dado que C1 ha sido eliminada al ser reconocida como implicante. La forma canónica contiene doce unos y cuatro ceros; en este procedimiento alternativo trataremos los ceros de la forma canónica en vez de los unos, como preconiza el algoritmo original.

PASO 2: Carga inicial de implicantes primos.

Se cargan cuatro, uno por cada cero de la forma canónica, que son:

Num.unos	Implicante primo
1	0001
2	0101
2	0110
3	1110

Lógicamente estos cuatro son justamente los que no se cargaron cuando se trataban los unos.

PASO 3: Fusión de *minterms* para obtener los Implicantes Primos

Ciertamente en este caso son *maxterms*, pero mantendré la terminología y me seguiré refiriendo a ellos como *minterms* para resaltar el hecho de que el funcionamiento del algoritmo en este procedimiento alternativo es exactamente el mismo que antes.

Comparaciones

(1) 0001 – (2) 0101 = 0–01
(1) 0001 – (2) 0110 = **NO** (hay más de una diferencia)

(2) 0101 – (3) 1110 = **NO**
(2) 0110 – (3) 1110 = –110

Todos los IPs, los cuatro, se han fusionado con algún otro, por lo que no hay que copiar ninguno adicional a la nueva tabla de implicantes primos, que queda:

Num.unos	Implicante primo
1	0–01
2	–110

Comparación (sólo hay una posible en la segunda ronda)

(1) 0–01 – (2) –110: **NO**

No hay más fusiones posibles. Ésta es, pues, la tabla final de implicantes primos. Con ella, en el **Paso 4** se genera la tabla bidimensional de implicantes primos y *minterms*, que es así de sencilla:

IP	0001	0101	1100	1110
0–01			X	X
–110	X	X		

No hay mucho más que hacer aquí: ambos implicantes primos son esenciales, por lo que el **Paso 5** selecciona ambos y a continuación en el **Paso 8** se genera la expresión resultante en base a estos dos únicos implicantes primos, 0–01 y –110. Esa expresión, según preconiza el **Paso 8**, es la siguiente:

N2*N4*C5+C3*C4*N5

Ahora bien, dado que hemos tratado los ceros de la forma canónica de la expresión original, **la fórmula obtenida es en realidad la de la expresión complementaria a la buscada**. Por lo tanto, para obtener la expresión real buscada basta con complementar la fórmula obtenida aplicando las Leyes de De Morgan. Una vez hecho esto el resultado, que ahora es un producto de sumas, es el siguiente:

(C2+C4+N5)*(N3+N4+C5)

Que tiene seis variables individuales en lugar de las ocho que tenía la expresión obtenida por el algoritmo tradicional, o bien siete tras aplicarle la propiedad distributiva. Es decir, **esta expresión es**, ahora sí**, la expresión mínima absoluta que resuelve la forma canónica 1011100111111101**. En este caso no se puede aplicar la distributiva para extraer factores comunes y se debe añadir, igual que antes, la variable implicante C1 sumando, con lo que la expresión final obtenida es:

C1+(C2+C4+N5)*(N3+N4+C5)

Si la comparamos con la fórmula obtenida por el procedimiento tradicional, que era: **C1+C2*N4+N5*(N3+N4)+C4*C5**, vemos que la recién obtenida tiene una variable individual menos. Y si operamos algebraicamente con una de las fórmulas llegaremos a obtener la otra, ambas son correctas y representan a la misma expresión.

El ejemplo anterior demuestra que hay casos en los que aplicar el algoritmo a los *maxterms* de la expresión, es decir, a los ceros de la forma canónica y luego negar la expresión resultante obtiene un mejor resultado, uno que contiene menos variables individuales que aplicándolo directamente a los *minterms*, los unos.

Con mucha dificultad he encontrado en la Red alguna información sobre la posibilidad de usar los *maxterms* en lugar de la forma universalmente aceptada de usar los *minterms* como se describe hasta ahora en este documento. Se trata de un artículo sobre diseño de circuitos electrónicos que cita la posibilidad de usar *maxterms* para obtener un Producto de Sumas, en vez de la habitual Suma de Productos, explicando que esto puede ser útil si la tecnología utilizada hace más eficiente el uso de puertas lógicas OR que el de las puertas AND: al usar *maxterms* resultan más puertas OR en el circuito que puertas AND, mientras que en el caso habitual en el circuito obtenido ocurre lo contrario.

Yo no sé casi nada sobre diseño de circuitos, por lo que no puedo opinar al respecto, pero repito que si lo que se desea es optimizar funciones lógicas en programas fuente, que es lo que a mí me interesa y de lo que sí que entiendo, lo crítico es obtener una función booleana con el menor número de variables (condiciones) posibles, y tanto da que haya más ORs que ANDs o viceversa.

Lo que no he encontrado en lugar alguno es que se cite el hecho de que tratando los *maxterms* se pueda llegar a una expresión de menor tamaño que si se tratan los *minterms,* y viceversa.

El caso es que queda demostrado que en ciertas ocasiones se obtiene una expresión mejor, con menos variables individuales representadas, tratando los *maxterm*s (los ceros de la forma canónica) y luego complementando la expresión resultante, que tratando los *minterms* (los unos) tal como se explica en la inmensa mayor parte de la literatura. Este hecho, el que tratar los ceros y aplicar las Leyes de De Morgan al resultado da en ocasiones mejores resultados que tratar los unos, no lo he encontrado así explicado en ninguna parte de la Red, por muchas búsquedas que he realizado. Ignoro si esta afirmación que hago aquí es realmente una primicia, aunque lo dudo; estoy prácticamente seguro de que en algún lugar alguien, quizás algún doctorando o investigador, ha llegado a la misma conclusión. Pero yo no le he encontrado.

Bien, en este punto de la investigación surgen naturalmente dos preguntas:

¿**Con cuánta frecuencia ocurre que dé mejores resultados**, es decir, una expresión de menor tamaño, **tratar los *maxterms* en lugar de los *minterms***?; y

¿**Es posible conocer a priori**, en base a la forma canónica, **si dará mejor resultado utilizar un método u otro**?

Dado que en principio no tenía ni idea de cómo podrían contestarse estas preguntas de forma teórica, la única manera que se me ocurrió para intentar responderlas era probando: ejecutar el algoritmo tanto con *minterms* como con *maxterms* con diferentes formas canónicas de distinto número de variables y estudiar el resultado comparando ambas soluciones, e intentar sacar conclusiones de estas pruebas. Para ello he construido diferentes programas de simulación que hacen exactamente esto: a partir de una determinada forma canónica se ejecuta el algoritmo de ambas formas, se comparan los resultados y se anota cuál de los dos métodos ha funcionado mejor en cada caso, o si ambos han dado como resultado fórmulas de igual tamaño. Y, desde luego, antes de nada he probado exhaustivamente el código del algoritmo en sí para asegurar que funciona correctamente en toda ocasión. Lo hace.

Los parámetros de las pruebas realizadas han sido los siguientes:

- He tratado formas canónicas de entre tres y diez variables individuales, es decir, entre 8 y 1024 bits en la forma canónica.
  Con dos variables diferentes los resultados son obvios, y mis medios materiales me imposibilitan tratar formas canónicas de más de 1.024 bits, y estas últimas con gran dificultad y consumo de recursos, tiempo de CPU, sobre todo.
  Espero que los resultados puedan extrapolarse con cierta fiabilidad a formas canónicas de mayor tamaño, pero desde luego que no puedo asegurar tal cosa.

- Para formas canónicas de tres o cuatro variables individuales (tamaños de 8 y 16 bits en la forma canónica) he realizado el proceso de comprobación para todas las posibles formas canónicas de dicha longitud, que son 254 en el caso de tres variables individuales y de 65.534 en el caso de cuatro (se excluyen las formas canónicas con todo ceros o todo unos, que dan como resultado cero y uno, respectivamente y no tienen ningún misterio).

Más allá de eso es para mí imposible realizar la comprobación de todas las posibles formas canónicas: con cinco variables y 32 bits en la forma canónica existen casi 4.300 millones de combinaciones posibles: imposible atacar la comprobación exhaustiva en ese caso, y no digamos para seis o más variables individuales, son cifras astronómicas.

- Consecuentemente, para formas canónicas de 32 bits en adelante he programado una simulación. El programa de simulación genera formas canónicas pseudoaleatorias, esparciendo los ceros y los unos en la forma canónica en base a un procedimiento razonablemente aleatorio, y realiza las comprobaciones con estos valores generados.
  La generación aleatoria tiene el inconveniente de que, por decirlo de algún modo, la expresión resultante "no tiene sentido". Me refiero a que cuando un programador escribe una cierta expresión lógica busca representar la realidad, algo que debe suceder en los datos para tomar uno u otro camino en la ejecución del programa; sin embargo, una forma canónica aleatoria no representa en principio nada "lógico", y esto ciertamente tiene consecuencias al evaluar la expresión.

- En todos los casos, dada una forma canónica obtenida de una u otra manera he ejecutado el algoritmo de ambas formas, incluyendo, desde luego, el Paso 7 (Obtención de la Solución Óptima) que aún no he llegado a definir, y he comprobado si es mejor uno u otro método contando las condiciones individuales contenidas en cada una de las dos expresiones devueltas, sea una Suma de Productos (*minterms*), sea un Producto de Sumas (*maxterms*).

- Además, para los tamaños más pequeños (hasta 6 variables diferentes) he realizado una simulación similar, pero esta vez aplicando a las expresiones resultantes la propiedad distributiva para sacar factores/sumandos comunes y reducir más aún las expresiones. La ejecución del proceso de extracción de factores o sumandos comunes es muy costosa cuando las expresiones son de gran tamaño, como las que se obtienen cuando la forma canónica generada aleatoriamente tiene, por ejemplo, 1.024 bits.

En total he tratado varios millones de formas canónicas de diferentes tamaños, como se muestra en la tabla siguiente:

Nº variables individuales	Nº bits de la forma canónica	Combinaciones probadas sin aplicar la distributiva	Combinaciones probadas aplicando la distributiva	Total de Combinaciones probadas
3	8	254	254	254
4	16	65.534	65.534	65.534
5	32	2.000.000	250.000	2.250.000
6	64	200.000	20.000	220.000
7	128	60.000	–	60.000
8	256	20.000	–	20.000
9	512	2.000	–	2.000
10	1024	250	–	250

Lógicamente, cuantas más variables (más bits en la forma canónica) más costoso es el proceso, y también es sensiblemente más costosa la búsqueda sistemática de factores/sumandos comunes; ésa es la razón de que cuanto mayor es el tamaño de la forma canónica, menor sea el número de simulaciones efectuadas, dado que mis medios materiales para la ejecución de estos procesos son, obviamente, limitados. Por ejemplo, en el caso de formas canónicas de 1.024 bits, la comprobación de ambos algoritmos para determinar cuál es la menor de las dos expresiones encontradas tarda en mi máquina de media cinco minutos por cada forma canónica... y eso sin intentar siquiera aplicar la propiedad distributiva, lo que podría llevar entre veinte y treinta

minutos adicionales por cada forma canónica. Efectivamente, usar una muestra de solamente 250 formas canónicas aleatorias para intentar determinar lo que les ocurre a las del orden de 10^{310} (2^{1024}) restantes es irrisorio, pero habrá que conformarse. Ir más allá es para mí completamente imposible. Y no digamos para formas canónicas de más de 1.024 bits.

En cualquier caso, del estudio de los diferentes resultados se pueden extraer conclusiones valiosas, que paso a detallar a continuación, intentando responder en lo posible a las dos preguntas anteriores.

Pregunta 1: ¿Con cuánta frecuencia ocurre que dé mejores resultados tratar los ceros, es decir, obtener una expresión de menor tamaño tratando los *maxterms* en lugar de los *minterms*?

Veamos: Para las formas canónicas de tres y cuatro variables (8 y 16 bits, respectivamente), donde se han evaluado la totalidad de formas canónicas posibles, y fijándonos en las expresiones devueltas por el algoritmo (sumas de productos o productos de sumas), **el número de formas canónicas para las que se obtiene mejor resultado tratando los unos *es exactamente el mismo* que aquellas para las que se obtiene mejor resultado tratando los ceros**.

He aquí los datos:

En formas canónicas de 8 bits (tres variables, 254 formas canónicas en total), en 60 casos (un 23,6%) se obtiene una expresión más pequeña tratando los unos, en otros 60 casos (otro 23,6%) se obtiene una expresión más pequeña tratando los ceros y en los 134 casos restantes (un 52,8%) ambos métodos dan expresiones de la misma longitud.

En cuanto a las formas canónicas de 16 bits (cuatro variables, 65.534 formas canónicas en total), en 23.304 casos se obtiene una expresión más pequeña tratando los unos, en otros exactamente 23.304 casos se obtiene una expresión más pequeña tratando los ceros y en los restantes 18.926 casos ambos métodos dan expresiones de la misma longitud. En porcentaje, en el 35,6% de los casos da mejor resultado tratar los unos; en otro 35,6%, tratar los ceros, y en el 28,8% restante ambos métodos dan expresiones del mismo tamaño.

Viendo estos datos, lo primero que a uno se le ocurre es que algo muy similar, mejor dicho, exactamente lo mismo va a pasar con todos los tamaños de forma canónica, pero ¿cómo podría yo confirmarlo, o no?

¿Qué ocurre en formas canónicas de mayor longitud, aquellas que se han investigado generando formas canónicas aleatorias? Pues que lógicamente el número de casos en que resulta mejor uno u otro método no coincide, pero la diferencia entre ellos es ciertamente pequeña y asumo que es perfectamente factible atribuirla a la variabilidad generada por el proceso de generación pseudoaleatoria de formas canónicas. Las cifras concretas de las simulaciones que he realizado de cada tamaño de forma canónica son las siguientes:

Formas canónicas de 32 bits: 930.364 casos donde es mejor tratar los unos; 952.825 casos donde es mejor tratar los ceros; y 116.811 donde da igual un método u otro. Los porcentajes de casos donde da mejor resultado un método u otro rondan ya el 47%.

Formas canónicas de 64 bits: 95.085 casos donde es mejor tratar los unos; 95.784 casos donde es mejor tratar los ceros; y 9.131 donde da igual un método u otro. Ahora los porcentajes de casos donde da mejor resultado bien un método, bien otro se acercan el 48%.

Formas canónicas de 128 bits: 28.345 casos donde es mejor tratar los unos; 29.690 casos donde es mejor tratar los ceros; y 1.965 donde da igual un método u otro.

Formas canónicas de 256 bits: 10.066 casos donde es mejor tratar los unos; 9.518 casos donde es mejor tratar los ceros; y 416 donde da igual un método u otro.

Formas canónicas de 512 bits: 930 casos donde es mejor tratar los unos; 1.043 casos donde es mejor tratar los ceros; y 27 donde da igual un método u otro.

Formas canónicas de 1024 bits: 93 casos donde es mejor tratar los unos; 153 casos donde es mejor tratar los ceros; y 4 donde da igual un método u otro.

Dado el bajo número de casos simulados para formas canónicas de 1.024 bits, sólo 250 de muchos billones de cuatrillones, es imposible sacar consecuencia alguna de estos datos, y de hecho lo mismo podemos decir de las formas canónicas de menor tamaño. Sin embargo, los datos sí que muestran una clara tendencia: en unas simulaciones ha dado mejores resultados tratar los ceros; en otras, los unos; y en los únicos casos en los que el tamaño de la forma canónica permite tratar todas las formas canónicas posibles, las de 8 ó 16 bits, se constata que hay un número idéntico de formas canónicas que dan mejor resultado tratando los unos que tratando los ceros.

Mi conjetura es, pues, la siguiente:

Para todo tamaño de forma canónica, el número de casos en los que la expresión mínima absoluta se obtiene tratando los unos (*minterms*) es exactamente el mismo que el número de casos en los que la expresión mínima se obtiene tratando los ceros (*maxterms*) y luego complementando la expresión obtenida aplicando las leyes de De Morgan.

¿Será esta afirmación algo más que una simple conjetura, habrá alguna forma de demostrar que esto es así, convirtiendo la conjetura en un teorema?

Sí que la hay. Vamos a ello:

Para convertir esta plausible conjetura en un teorema hay que demostrar que para todo tamaño de forma canónica ocurre este hecho: que el número de formas canónicas en que resulta óptimo utilizar un método es idéntico al de formas canónicas en que es óptimo utilizar el otro. Sabemos que con tamaños de forma canónica de 4, 8 y 16 bits esta conjetura se cumple, constatándolo por fuerza bruta, es decir, probando la totalidad de formas canónicas posibles de estas longitudes. Sólo queda extender la prueba a formas canónicas de mayor tamaño, donde es imposible utilizar la fuerza bruta debido al gigantesco número de formas canónicas posibles.

Vamos a hacerlo en base a una serie de Lemas que iré demostrando hasta llegar a la demostración final. Algunos son evidentes, pero no quiero dar nada por sentado y ser lo más riguroso posible.

En primer lugar definiré el concepto de "*forma canónica espejo*" de una forma canónica dada, de la forma:

"**Dada una determinada forma canónica *FC*, existe una y sólo una forma canónica espejo de ella, *FE*, de su misma longitud, tal que todos los bits de *FE* tienen los valores contrarios a los de la forma canónica *FC* en su misma posición**".

Por ejemplo, dada la forma canónica *FC*: 1101010001010101, existe una y sólo una forma canónica espejo *FE* de ella y de igual longitud: 0010101110101010. Y desde luego, a su vez *FC* es la forma canónica espejo de *FE*. Obviamente, esta definición implica que *FE* es la forma canónica de la expresión complementaria a la de *FC*, y viceversa.

LEMA 1: Toda forma canónica tiene una única forma canónica espejo.

Como cada forma canónica es distinta a todas las demás de su misma longitud, invirtiendo los valores de todos sus bits se llega a otra forma canónica de las posibles que, a su vez, es única. Por lo tanto, todas las formas canónicas de una cierta longitud se organizan en parejas, donde cada una de ellas es espejo, es decir, complementaria, de la otra.

LEMA 2: Dada una determinada longitud *n* de formas canónicas, donde el número N de formas canónicas diferentes es $N=2^n$, existen N/2 parejas de formas canónicas donde una es espejo de la otra, y viceversa.

Dado que toda forma canónica de las 2^n posibles tiene una única forma canónica espejo y viceversa, el número total de parejas existentes es el número N dividido por el número de componentes de cada pareja, 2, y por tanto N/2, es decir $2^n/2$.

Habiendo quedado bien establecida la definición de las parejas de formas canónicas espejo, fijémonos ahora en cómo trata el algoritmo de Quine-McClusky a cada una de estas formas canónicas espejo. Observaremos qué ocurre tanto cuando para cada una de ellas se tratan los unos como cuando se tratan los ceros.

Recordemos en este punto que en el Paso 1 del algoritmo QM se extraen las posibles variables implicantes que pudiera tener la forma canónica dada.

LEMA 3: Si una cierta forma canónica contiene una o varias variables implicantes, entonces su forma canónica espejo contiene también el mismo número de variables implicantes.

De hecho, las de la forma espejo son las mismas variables que las de la inicial, pero complementadas, en su mismo orden y afectadas por el operador contrario: si una variable implicante C*x* está afectada por el operador '+', en la forma espejo estará complementada, es decir, será N*x*, y estará afectada por el operador contrario, '*'.

Si una determinada configuración de unos en una de las formas determina que existe una variable implicante sumando, entonces en la forma espejo existe una configuración idéntica de ceros en sus mismas posiciones, y por lo tanto la misma variable, pero complementada, es implicante, pero esta vez multiplicando. Y viceversa, si una determinada configuración de ceros en una de las formas determina que existe una variable implicante multiplicando, entonces en la forma espejo existe una configuración idéntica de unos en sus mismas posiciones, y así la misma variable, complementada, es implicando, pero ahora sumando.

Veamos un ejemplo; en concreto usaremos el mismo que hemos usado hasta ahora, de la forma canónica 10111001111111011111111111111111. Sabemos que C1 era una variable implicante, sumando, dado que todos sus unos eran unos también en la forma canónica:

Forma canónica: 10111001111111011111111111111111
Tabla de verdad de C1: 0000000000000000**1111111111111111**

Una vez extraída esta variable implicante C1, sumando, la forma canónica resultante quedaba: 1011100111111101.

Y por otro lado, la forma canónica espejo de 10111001111111011111111111111111 es, claro está: 01000110000000100000000000000000.

En esta forma canónica espejo podemos ahora comprobar que, al compararla con la tabla de verdad de N1, la complementaria de C1, todos sus ceros coinciden con ceros en la forma espejo:

Forma canónica espejo: 0100011000000010000000000000000
Tabla de verdad de N1: 1111111111111111**0000000000000000**

Por lo tanto N1, complementaria de C1, es implicante, pero en este caso multiplicando, de la forma espejo de la original. Y la forma canónica resultante tras extraer dicha variable implicante N1 es la siguiente: 0100011000000010.

LEMA 4: Tras la extracción de las variables implicantes de una forma canónica y de su forma espejo en el Paso 1 del algoritmo QM, las dos formas canónicas resultantes tras la extracción siguen siendo la una espejo de la otra.

La eliminación del mismo número de bits en ambas formas canónicas de la pareja de formas espejo, y en las mismas posiciones, deja una serie de bits que no son modificados y en sus mismas posiciones relativas, y por lo tanto continúan siendo unos los contrarios de los otros. Lo podemos ver en el caso anterior: las formas canónicas resultantes de la extracción en las dos formas canónicas espejo son: 1011100111111101 y
0100011000000010.

Ambas continúan, pues, formando una pareja de formas canónicas espejo.

LEMA 5: La expresión resultante de aplicar el algoritmo de Quine-McClusky a una forma canónica determinada tratando los unos tiene idéntica longitud que la de la expresión obtenida por dicho algoritmo aplicado a su forma canónica espejo, pero tratando los ceros y complementando, vía las Leyes de De Morgan, la expresión resultante.

Y viceversa, la expresión resultante de aplicar el algoritmo de Quine-McClusky a una determinada forma canónica tratando los ceros y complementando la expresión resultante tiene idéntica longitud que la de la expresión que obtiene dicho algoritmo aplicado a la forma canónica espejo, pero tratando los unos.

Fijémonos en cómo se produce la carga inicial de *minterms* en la tabla de implicantes primos, o propiamente *maxterms*, si se tratan los ceros. Sean las formas canónicas *FC* y *FE* que forman una pareja de formas espejo, que van a ser objeto de tratamiento por el algoritmo QM. Tomemos primero a la forma canónica *FC* tratando los unos. El Paso 2 del algoritmo carga en la tabla de IPs todos los *minterms* cuyo valor en dicha forma *FC* es un uno.

Comparemos ahora con lo que el propio Paso 2 carga en la tabla de IPs si tomamos ahora la forma canónica *FE*, pero tratando los ceros. Se cargarían en la tabla de IPs todos los *maxterms* cuyo valor en dicha forma *FC* es un cero. Comparemos ahora el contenido de ambas tablas de IPs tras esta carga inicial en uno y otro caso.

¡Son idénticas!

En efecto, al formar las formas canónicas *FC* y *FE* una pareja de formas espejo, todos los unos existentes en la forma *FC* son ceros en las mismas posiciones de la forma *FE*, y viceversa. Por lo tanto, tratando los unos con la forma canónica *FC* se cargan todos los *minterms* con valor uno, que son exactamente los mismos que tienen valor cero en la forma canónica *FE*. Pero en esta forma canónica *FE* se están tratando precisamente los ceros, los *maxterms*, por lo que los valores que en definitiva se cargan inicialmente en la tabla de IPs son los mismos.

Todo el resto del algoritmo, los Pasos 3 a 8, es idéntico en ambos casos, dado que sus datos de entrada, la tabla de IPs inicialmente cargada en dicho Paso 2, son los mismos. El único punto en que varía el algoritmo en ambos casos es que en el paso final, una vez obtenida la expresión resultante y antes de añadir las variables implicantes, en el caso de tratar los ceros la expresión debe ser complementada usando las Leyes de De Morgan, mientras que si se tratan los unos este paso no se realiza.

Sin embargo, la ejecución de dicha complementación o negación de la expresión resultante no varía el número de variables de la expresión: todas las variables son sustituidas por sus complementarias, C1 por N1, etc., y los operadores se intercambian, donde hay un '+' se cambia por '*' y viceversa. Puede ser necesario añadir paréntesis a la expresión, pero esto no altera el número de variables de la expresión.

Luego las longitudes de las expresiones así obtenidas son idénticas.

LEMA 6: Si en una determinada pareja de formas canónicas espejo {*FC*, *FE*} el algoritmo QM aplicado a la forma *FC* obtiene la solución mínima tratando los unos, el algoritmo aplicado a su forma canónica espejo *FE* obtendrá la solución mínima tratando los ceros, y viceversa.

Es consecuencia inmediata del LEMA 5. Las expresiones obtenidas por el algoritmo QM tratando los unos en una forma canónica y tratando los ceros en su forma canónica espejo son siempre de la misma longitud, y viceversa.

Por tanto, si en una de estas dos formas canónicas se obtiene mejor resultado tratando, por ejemplo, los unos, en su forma canónica espejo se obtendrán los mejores resultados tratando los ceros, y viceversa.

Y, por fin, en aquellas formas canónicas en las que se obtengan expresiones de la misma longitud bien aplicando el algoritmo a los unos, bien a los ceros, también ocurrirá lo mismo en sus respectivas formas canónicas espejo, dado que las longitudes de las expresiones están ligadas dos a dos: tratando los ceros en una y tratando los unos en la otra.

Veamos un ejemplo. Sea la pareja de formas canónicas espejo siguiente, de longitud 8 y, por tanto, de tres variables individuales C1, C2 y C3.

FC: 10011000 y
FE: 01100111.

Cada una es espejo de la otra. No hay variables implicantes, en ninguna de las dos. Recordemos que si en una de ellas se hubieran detectado una o varias variables implicantes, en la otra se habrían detectado el mismo número de ellas, debido al LEMA 3.

Apliquemos en primer lugar el algoritmo QM tratando los unos a la forma *FC*. La carga inicial de *minterms* en la tabla de IPs cargaría los siguientes: {000, 011 y 100}. A partir de ahí el algoritmo se ejecuta normalmente y el resultado es la expresión: N1*C2*C3+N2*N3, con cinco variables individuales en ella.

Si aplicamos ahora el algoritmo QM a la forma *FE*, pero tratando los ceros, vemos que los *maxterms* cargados inicialmente en la tabla de IPs son exactamente los mismos que en el caso anterior: {000, 011, 100}. Por lo tanto, el algoritmo devolverá exactamente la misma expresión N1*C2*C3+N2*N3, que debe en este caso ser complementada para obtener la expresión definitiva: (C1+N2+N3)*(C2+C3), también con cinco variables individuales y, por lo tanto, del mismo tamaño que tratando los unos en la forma espejo.

De forma similar, si aplicamos ahora el algoritmo QM tratando ceros a la forma *FC*, cargará inicialmente en la tabla de IPS los cinco *maxterms* correspondientes a los cinco ceros de *FC*, que son: {001, 010, 101, 110, 111}. Ejecutando ahora el algoritmo QM, obtiene la fórmula siguiente, ya complementada mediante las Leyes de De Morgan: (C2+N3)*(N2+C3)*(N1+N2), con seis variables individuales. Por tanto, es óptimo en esta forma canónica *FC* usar el algoritmo QM tratando los unos.

Aplicando ahora el algoritmo QM a la forma *FE*, pero esta vez tratando los unos, la carga inicial de la tabla de IPS termina con los mismos cinco *minterms* del caso anterior en la tabla, y tras ejecutar el algoritmo se obtiene la expresión: N2*C3+C2*N3+C1*C2, de seis variables, y por ello en el caso de la forma canónica *FE* el resultado óptimo se obtiene tratando los ceros en el algoritmo QM.

Es decir, en el caso de la pareja de formas espejo *FC* y *FE* del ejemplo, en una de las dos, *FC*, la expresión mínima se obtiene tratando los unos, mientras que en la otra, *FE*, el resultado óptimo se obtiene tratando el valor contrario, los ceros. Circunstancia que ocurre en toda pareja de formas canónicas espejo de cualquier tamaño, excepto que con ambos métodos (tratar unos o ceros) se obtengan expresiones del mismo tamaño, en cuyo caso es irrelevante usar uno u otro método para obtener la expresión mínima correspondiente a las dos formas canónicas espejo que forman la pareja.

TEOREMA: Hay un número idéntico de formas canónicas de una determinada longitud en las que el algoritmo QM obtiene la expresión mínima tratando los unos que formas canónicas de la misma longitud en las que el algoritmo QM obtiene la expresión mínima tratando los ceros.

Es consecuencia inmediata del LEMA 6, que garantiza que en toda pareja de formas canónicas espejo, si en una de ellas resulta óptimo tratar los unos, en la otra resultará siempre óptimo tratar el valor contrario, los ceros, y viceversa, y del LEMA 2, que indica que para formas canónicas de *n* bits de longitud existen N/2 parejas de formas canónicas espejo diferentes, siendo N el número total de formas canónicas posibles para dicha longitud *n*, es decir, $N=2^n$.

Cada pareja de formas canónicas espejo puede ser de dos tipos: 1) en la pareja es irrelevante qué método utilizar en el algoritmo QM, debido a que ambos métodos, tratar unos o ceros, obtienen expresiones del mismo tamaño; y 2) en una de las dos formas canónicas espejo es óptimo tratar los ceros y en la otra lo es tratar los unos.

Y por tanto, **el número total de formas canónicas de longitud *n* en las que la solución mínima se obtiene tratando los ceros (*maxterms*) y complementando la expresión final mediante las Leyes de De Morgan es idéntico al número total de formas canónicas de dicha longitud *n* en las que la solución mínima se obtiene tratando los unos (*minterms*)**.

QED

Hasta aquí, todo esto es fijándose en la expresión en bruto obtenida por el algoritmo QM, sea ésta una suma de productos, sea un producto de sumas. Es obvio que en una parte significativa de los casos esta expresión se puede reducir aplicando la propiedad distributiva para sacar factores/sumandos comunes. Entonces, ¿qué ocurre cuando se aplica la distributiva, cambian algo los resultados?

Bien, veamos ahora cuáles son los resultados de aplicar la propiedad distributiva en los cuatro tamaños de forma canónica que he estudiado (8, 16, 32 y 64 bits):

Formas canónicas de 8 bits (tres variables): Todos los 254 casos dan ahora expresiones de exactamente el mismo tamaño usando uno u otro método.

Formas canónicas de 16 bits (cuatro variables): Ahora la cifra de casos en que resulta mejor aplicar uno u otro método ya no es exactamente la misma: 15.527 casos donde es mejor tratar los unos; 15.590 casos donde es mejor tratar los ceros; y 34.417 donde da igual un método u otro. Han aumentado significativamente (se han más que cuadruplicado) los casos en los que ambos métodos dan expresiones del mismo tamaño, pero ahora ya no son exactamente iguales los casos en que da mejor resultado un método o el otro. La diferencia es pequeña, 15.527 vs. 15.590, pero ya no son iguales.

En vista del TEOREMA que acabo de definir esto no es posible, porque aplicar la propiedad distributiva a expresiones que ya hemos visto que son complementarias unas de otras debería obtener siempre expresiones del mismo tamaño, y los datos indican que no siempre lo hace, y eso no es teóricamente posible. Pero en el mundo real ocurre. Al menos en *mi* mundo real. La explicación de que se dé esta diferencia, explicación que he contrastado a base de reproducir de forma manual algún caso concreto, es la siguiente:

Todo algoritmo que trate de reducir el tamaño de una expresión lógica aplicando la propiedad distributiva para sacar factor/sumando común a ciertos términos tiene que, tras analizar la expresión, elegir un determinado término de todos los posibles para extraerlo el primero de la fórmula, ignorando el resto, y esto debe hacerlo iterativamente tantas veces como pueda mientras queden términos repetidos en la fórmula. En el ejemplo citado en el capítulo anterior, recordemos, se podía elegir N4 o N5 como variable para extraerla como factor común, pero no a ambas.

Pues bien, por muy sofisticado que sea el algoritmo elegido, que tenga en cuenta tanto el número de veces que se encuentra en la fórmula cada término repetido como su tamaño, y seleccione el más idóneo según sus parámetros, y por muy sofisticados que sean dichos parámetros, siempre existirán fórmulas que se hubieran podido reducir más escogiendo otros términos o haciéndolo en otro orden. Y esto es exactamente lo que ocurre aquí: esa diferencia de 63 casos en el número de casos (15.527 vs 15.590) se produce precisamente debido a este sesgo del algoritmo de aplicación de la propiedad distributiva.

Cierto, seguro que se puede escribir un algoritmo mejor, pero yo no sé hacerlo. Bueno, sí que sé, tratando todas las posibles formas de extraer factores/sumandos comunes y quedándose con el que mejor resultado dé, pero yo no pienso programar esa monstruosidad. Sigamos.

Formas canónicas de 32 bits (cinco variables): Hay 99.609 casos donde es mejor tratar los unos; 101.677 donde es mejor tratar los ceros; y 48.714 casos donde las expresiones obtenidas por uno u otro método son de idéntico tamaño.

Formas canónicas de 64 bits (seis variables): Hay 8.628 casos donde es mejor tratar los unos; 9.244 en los que es mejor tratar los ceros; y 2.128 donde da igual usar uno u otro método.

En resumen, se puede extender el TEOREMA antes definido si las expresiones obtenidas son reducidas mediante la aplicación de la propiedad distributiva. Así se pueden obtener fórmulas de menor tamaño que las obtenidas directamente por el algoritmo, si ése fuera el objetivo de la aplicación del algoritmo. Lo que sí ocurre siempre tras la aplicación de la distributiva es que aumentan considerablemente los casos en que ambos métodos, tratar unos y tratar ceros, obtienen expresiones del mismo tamaño.

En cualquier caso, y según los datos de las simulaciones realizadas, el porcentaje de formas canónicas en que es indiferente usar uno u otro método se reduce conforme el tamaño de las formas canónicas aumenta: de un 52,8% de las formas canónicas de 8 bits de tamaño, se pasa a un 28,8% en las de 16 bits; a un 5,8% en las de 32 bits; un 4,5% en las de 64 bits; un 3,2% en las de 128 bits; a un magro 2,1% en las de 256 bits, etc. Y esto implica que conforme aumenta el tamaño de la forma canónica la probabilidad de que el resultado óptimo se consiga tratando bien los unos, bien los ceros, se aproxima al 100%.

Es decir, en base a los datos conseguidos mediante las diferentes simulaciones podemos asegurar que **aplicando el algoritmo de Quine-McClusky tal como se define en la literatura hay una alta probabilidad, cercana al 50% para formas canónicas de una cierta longitud, de que la expresión obtenida no sea la mínima absoluta correspondiente a la forma canónica dada**.

La reflexión inmediata tras todo este discurso es que para asegurar que se obtiene en todo caso la expresión mínima correspondiente a una determinada forma canónica es necesario ejecutar el algoritmo QM dos veces, una tratando los unos, la forma tradicional, y otra tratando los ceros y complementando la expresión resultante mediante las Leyes de De Morgan, para calcular el tamaño de cada una de las expresiones así obtenidas y quedarse con la de menor tamaño.

Pregunta 2: ¿Es posible conocer a priori, en base a la forma canónica, **si dará mejor resultado utilizar un método u otro**?

La respuesta corta es: **No**. O mejor, **casi seguro que no**. O mejor aún, yo no he encontrado la forma de hacerlo.

Por más suposiciones y pruebas que he hecho (que la forma canónica tenga más ceros que unos o viceversa, que se den en ella ciertas configuraciones de ceros y unos, etc.) no he encontrado ningún criterio que pueda determinar a priori si va a dar mejor resultado uno u otro método. Como mucho se puede seleccionar probabilísticamente qué método va a dar mejor resultado en base a observar algunos millones de casos, como yo he hecho.

Veamos un ejemplo de lo que ocurre en el caso de las formas canónicas de 16 bits (es decir, de cuatro variables), de las que he tratado todas y cada una de las 65.534 formas canónicas posibles, cuando se discriminan los resultados según el número de ceros y unos contenidos en la forma canónica:

Expresión directamente devuelta por el algoritmo (como suma de productos o producto de sumas)				
	Característica de la forma canónica			
Resultado	**Más unos que ceros**	**Más ceros que unos**	**Igual número**	**TOTALES**
Mejor tratar Unos	13.288	7.664	2.352	23.304
Mejor tratar Ceros	7.664	13.288	2.352	23.304
Mismo resultado	5.380	5.380	8.166	18.926
TOTALES	26.332	26.332	12.870	65.534

Dadas estas cifras, la mejor estrategia consiste en:
- Si la forma canónica tiene más unos que ceros, o los mismos, entonces tratar los **unos**;
- Si la forma canónica tiene más ceros que unos, entonces tratar los **ceros**.

Comprobadlo, si lo deseáis. Esta estrategia acierta en el 73,02% de las ocasiones, y por lo tanto falla en el restante 26,98% de las veces. Es más corto calcular la probabilidad de fallo: 7.664+7.664+2.352=17.680 de 65.534, un 26,98%. Que no deja de ser una alta probabilidad de fallo, me parece a mí, como para grabar en piedra la estrategia.

¿Qué pasaría si a estas mismas expresiones en bruto entregadas por el algoritmo en cualquiera de sus dos formas, tratar unos o ceros, les aplicamos la propiedad distributiva para reducir su tamaño sacando factores/sumandos comunes? Veamos:

Expresión resultante tras aplicar la propiedad distributiva				
	Característica de la forma canónica			
Resultado	**Más unos que ceros**	**Más ceros que unos**	**Igual número**	**TOTALES**
Mejor Unos	8.111	4.668	2.748	15.527
Mejor Ceros	4.687	8.100	2.803	15.590
Da igual	13.534	13.564	7.319	34.417
TOTALES	26.332	26.332	12.870	65.534

La mejor estrategia ahora es casi la misma que en el caso anterior, pero ahora la tasa de acierto es mejor, consecuencia directa de que el número de casos en los que los resultados de aplicar ambos métodos obtienen ahora expresiones del mismo tamaño:

- Si la forma canónica tiene más unos que ceros, entonces tratar los **unos**;
- Si la forma canónica tiene más ceros que unos, o los mismos, entonces tratar los **ceros**.

Esta estrategia falla en obtener la expresión mínima en un 18,47% de las ocasiones (4.687+4.668+2.748=12.103 casos de 65.534), y por tanto acierta en un 81,53% de las veces.

Es decir, aplicar la distributiva mejora la tasa de acierto de la estrategia de elegir uno u otro método, pero conforme va aumentando el tamaño de las formas canónicas, la probabilidad de acertar eligiendo de antemano uno u otro método va disminuyendo en cualquier caso hasta acercarse al 50%, lo que básicamente es lo mismo que elegir uno u otro método de forma aleatoria, o incluso mejor, elegir siempre el mismo.

Así que si lo que se desea es **obtener siempre la expresión mínima** correspondiente a una determinada forma canónica no queda más remedio que **ejecutar dos veces el algoritmo conforme a ambos métodos**, tratando los unos y tratando los ceros, **y quedarse con la expresión que resulte de menor tamaño**. Y, desde luego, aplicar la propiedad distributiva a la expresión obtenida para extraer cuantos factores/sumandos comunes sea posible.

Terminada la disquisición sobre la conveniencia de usar un método (tratar los *minterms*) u otro (tratar los *maxterms*), es el por fin momento de definir el paso restante del algoritmo, ése que había dejado para más adelante, el **Paso 7**, **Obtener la Solución Óptima**, el que de verdad precisa cantidades astronómicas de recursos informáticos para ejecutarse. El problema de minimizar expresiones lógicas es un problema *NP-completo*, como ya dije, y es precisamente en este Paso 7 donde esa *NP-Completitud* se muestra en todo su esplendor.

La búsqueda de la expresión mínima

Hasta este momento he explicado cómo es el algoritmo de Quine-McClusky y de qué modo se puede incrementar su eficacia, proporcionando siempre y en toda ocasión la expresión mínima correspondiente a la forma canónica dada, por el método de calcular la fórmula resultante tratando, por un lado, los unos, los *minterms*, y por el otro tratando los ceros, los *maxterms*, y complementando la expresión resultante, para quedarse por fin con la expresión más pequeña de las dos.

Pero esa definición todavía no está completa: falta por describir el penúltimo paso del algoritmo que, dadas las características de los ejemplos utilizados en el documento, hasta ahora no ha hecho falta, pero en el caso general ya lo creo que hace falta. Vamos con ese Paso 7.

Paso 7 – Búsqueda de la Solución Óptima

Recordemos que tras la ejecución del paso 6 (en realidad, del paso conjunto 5-6) puede que resten en la tabla de cobertura algunos (o muchos) implicantes primos que cubren un determinado número de columnas (*minterms*), al no ser ninguno de ellos esencial ni estar dominado por otro. Es normal que sí que resten una serie de IPs en la tabla, que hay que tratar para obtener la función mínima buscada, seleccionando los implicantes primos adecuados e incluyéndolos en la tabla de IPs seleccionados.

Antes de entrar en harina, ¿cómo son estos implicantes primos restantes, qué características tiene la tabla de cobertura resultante en este momento? En base al resultado de los pasos anteriores podemos hacer algunas afirmaciones útiles sobre ella:

1) Todas las columnas (*minterms*) de la tabla tienen al menos dos equis.
   Efectivamente, si alguna columna tuviera una sola equis, entonces el IP correspondiente sería esencial, habría sido seleccionado en el Paso 5 y ya no estaría en la tabla.

2) Todos los IPs (filas) de la tabla tienen al menos dos equis.
   Si hubiera alguno con una sola equis, y dado que según el punto anterior todas las columnas tienen al menos dos equis marcadas, entonces ese IP sería redundante con, al menos, el otro IP que tiene también marcada esa misma columna, y por tanto habría sido eliminado por el Paso 6.

3) Hay al menos tres IPs (filas) en la tabla de cobertura.
   Si quedaran solamente dos IPs, entonces o son iguales o uno de ellos es redundante con el otro, debido a los puntos 1 y 2 anteriores, y uno de ellos, el redundante, habría sido eliminado por el Paso 6.

4) Hay al menos tres *minterms* (columnas) en la tabla.
   Si hubiera sólo dos, entonces todos los IPS serían iguales, dado que todos los IPs tienen que tener al menos dos equis cada uno.

De esta forma la dimensión mínima de la tabla de cobertura en este punto del algoritmo QM es de 3 filas (IPs) y 3 columnas (*minterms*), y sería algo parecido a esto:

IP	m1	m2	m3
IP1		X	X
IP2	X	X	
IP3	X		X

Antes de comenzar a debatir las complejidades de este Paso 7 voy a definir un ejemplo que ayudará a fijar las ideas. Se trata de la forma canónica **1101111111111011**, de 16 bits y, por lo tanto, con cuatro variables involucradas, que son en este caso C1, C2, C3 y C4. La forma canónica tiene catorce unos y dos ceros; me centraré en lo que sigue en los resultados del algoritmo QM original, es decir, tratando los *minterms* (los unos). Advierto aquí que si hubiera elegido en su lugar la forma canónica *espejo*, la 0010000000000100, hubiera llegado a idénticas conclusiones tratando los *maxterms*, los ceros. Pero aquí lo mejor para nuestros intereses es tratar los unos, los *minterms*, tal como preconiza el algoritmo de Quine-McClusky habitual.

Vamos a aplicar el algoritmo a esta forma canónica.

En el Paso 1 no se detecta ninguna variable implicante. En consecuencia no se cambia la forma canónica para la ejecución del resto de pasos, ni se incluye ninguna variable en la tabla de variables implicantes, que por lo tanto queda vacía.

Los *minterms* originales que se cargan en la tabla de IPs en el Paso 2 son, pues, catorce:
0000, 0001, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1110 y 1111.
Sólo faltan el 0010 y el 1101, que corresponden a los dos únicos ceros de la forma canónica.

Ahora se ejecutaría el Paso 3, Cálculo de los Implicantes Primos. Vais a tener que creerme, o mejor, al programa que calcula estas cosas: este paso deja finalmente doce IPs que, una vez cargados en la tabla de cobertura según preconiza el Paso 4, quedan así:

	***Minterms* de la expresión**													
IPs	**0000**	**0001**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1110**	**1111**
--00	X			X				X				X		
--11			X				X				X			X
-0-1		X	X						X		X			
-00-	X	X						X	X					
-1-0				X		X						X	X	
-11-						X	X						X	X
0--1		X	X		X		X							
0-0-	X	X		X	X									
01--				X	X	X	X							
1--0								X		X		X	X	
1-1-										X	X		X	X
10--								X	X	X	X			

Bien, sobre esta tabla de cobertura de 12 IPs (filas) y 14 *minterms* (columnas) se procesa el Paso 5, Selección de Implicantes Primos Esenciales... ¡No hay ninguno! Y como consecuencia el Paso 6 tampoco encuentra ningún IP redundante. De hecho, si no se ha seleccionado y eliminado ningún implicante primo esencial, y dado el método de construcción de la tabla, es imposible que haya implicantes primos redundantes. En una palabra, ésta es definitivamente la tabla de cobertura que tiene que tratar este Paso 7.

El objetivo de dicho Paso 7 es encontrar una combinación de IPs tales que, en primer lugar, queden cubiertas todas las columnas de la tabla (necesario para asegurar que la fórmula encontrada es correcta, pues cubre todos los *minterms* de la forma canónica) y, luego, de todas las posibles combinaciones de IPs que cumplen esta condición, encontrar la que contenga un menor número de variables individuales, es decir, donde la suma del número de ceros y unos de los IPs de la combinación sea mínima, la menor de todas ellas.

A cualquiera se le ocurre un método infalible para encontrar esta solución: se generan todas las combinaciones de IPs posibles y se verifica para cada una de ellas si efectivamente todas las columnas han sido cubiertas por la combinación. Si quedan columnas por cubrir se descarta la combinación, pero si todas las columnas están cubiertas se cuentan las variables totales de dicha solución; nos quedamos con la combinación que menor número de variables tiene, y listo.

Un método éste, el de **fuerza bruta**, sencillo y eficaz... siempre que el número de IPs de la tabla esté en un cierto rango. En el caso del ejemplo son 12 IPs, por lo tanto las combinaciones a probar son 4.095 (2^{12}-1), desde 000000000001 hasta 111111111111, donde un 1 en la posición *i-ésima* significa que el IP correspondiente, el que a su vez está en la fila *i-ésima* de la tabla de cobertura, forma parte de la combinación, y 0, que no forma parte: el número de combinaciones posibles con *n* IPs es, obviamente, 2^n, y como la combinación de todo ceros, es decir, no seleccionar ningún IP, no tiene sentido, el número de combinaciones a probar es de 2^n-1.

Cierto que hay algunos trucos que permiten que ciertas combinaciones puedan ser ignoradas de antemano, como las que tienen un único uno (es imposible que con un solo IP se cubran todas las columnas; si eso pasara no habríamos llegado a este Paso 7), o bien aquellas que tengan más unos que la mejor combinación seleccionada hasta el momento, etc., pero esto no varía en demasía el número de combinaciones a probar, que crece exponencialmente con el número de IPs de la tabla.

En nuestro ejemplo el coste de realizar 4.095 comprobaciones es, seguramente, asumible, pero ¿qué ocurre si en la tabla de cobertura hay, por ejemplo, 100 IPs? Pues que comprobar 2^{100} combinaciones, es decir, del orden de 10^{30}, está completamente fuera del alcance no sólo de mí, sino de cualquier ordenador que yo conozca. Esta cifra de 100 IPs es bastante habitual en cuanto la forma canónica tiene 256 o más bits; para formas canónicas de 1.024 bits lo normal es que la tabla de cobertura tenga en este punto al menos 150 IPs y 120 columnas; yo he detectado alguna forma canónica de 1.024 bits que genera una gigantesca tabla de cobertura de 556 filas y 464 columnas. Tratar esta monstruosidad por fuerza bruta es algo completamente fuera de las posibilidades de cálculo de todo el mundo. Y eso con solamente 1.024 bits en la forma canónica; imaginad lo que ocurre con formas canónicas mayores.

Por lo tanto, el método de fuerza bruta queda descartado como método factible en cuanto la tabla tiene más allá de quizás 20 ó 25 filas. Y como consecuencia hay que buscar métodos alternativos que encuentren la combinación ideal en un tiempo razonable, o por lo menos que sea en tiempo polinómico, no exponencial. Y en este caso la "*combinación ideal*" no tiene por qué ser la mínima absoluta, sino una que se le acerque lo suficiente: ya hemos visto que asegurar que se ha encontrado la solución mínima absoluta es prácticamente imposible en cuanto el número de IPs de la tabla supera una cierta, y relativamente modesta, cifra.

En la literatura sobre el algoritmo de Quine-McClusky que he revisado siempre se citan en este punto dos posibles métodos, aparte de la fuerza bruta, para resolver el problema de encontrar la combinación de implicantes primos que, cubriendo todas las columnas (los *minterms*), sea mínima, en el sentido de o bien necesitar la menor cantidad posible de IPs, o bien de involucrar la menor cantidad posible de variables individuales en la solución. Estos dos métodos son el *branch&bound* y el método de Petrick. Veamos cómo son cada uno de ellos.

El algoritmo branch&bound (ramificación y poda, en español) es un algoritmo de programación lineal entera, una variante del backtracking donde a partir de una combinación inicial válida se va construyendo un árbol de soluciones en el que cada rama conduce a una posible solución más evolucionada que la actual. El algoritmo elimina en cada paso aquellas ramas que se alejan de la solución óptima, para progresar sólo en aquellos nodos que van paulatinamente mejorando la solución, hasta llegar a un punto donde o no es posible mejorarla más o bien se alcanza un cierto límite de recursos prefijado.

Es una técnica similar a la *poda alfa-beta* tan utilizada en los programas que juegan al ajedrez, donde primero se elabora el árbol de jugadas posibles, se evalúan y entonces se podan del árbol las jugadas malas, para continuar expandiendo el árbol solamente con las jugadas prometedoras hasta alcanzar el límite fijado.

El algoritmo *branch&bound* corre en tiempo polinómico y, al igual que todos los algoritmos de programación lineal, no siempre devuelve la solución óptima, sino una que es "lo suficientemente óptima". Eso sí, por mucho que necesite un tiempo polinómico, cuando la tabla de cobertura tiene muchos implicantes primos el tiempo necesario para resolverlo se incrementa mucho. Muchísimo. Y además, cuanto más grande es dicha tabla de cobertura más dificultades tiene para encontrar la solución óptima.

Dado que hay mucha información en la Red sobre este tipo de algoritmos de programación lineal, herederos en su mayoría del venerable Simplex, no voy a incidir más en él, dado que, como se verá, hay otros métodos más eficientes y sencillos para obtener resultados similares.

En cuanto al **método de Petrick**, se trata en realidad de un algoritmo cuya función básica es convertir una función booleana expresada como un producto de sumas, es decir, en Forma Normal Conjuntiva, en una equivalente expresada como una suma de productos, y por tanto en Forma Normal Disyuntiva. Nada más. Y nada menos.

Se basa en una inteligente interpretación del contenido de la tabla de cobertura de implicantes primos, que voy a explicar en detalle: a mí me ha costado lo suyo entender bien los intríngulis del método, así que intentaré que a quienes lean estas líneas no les cueste tanto como a mí me costó. Veamos primero el artificio booleano en que se basa, es muy interesante.

Pongamos antes que nada nombres a las diferentes filas y columnas de la tabla de cobertura. Denominaremos las columnas (los *minterms*) como M1, M2... M*n*. En la tabla del ejemplo antes citado hay 14 columnas, luego sus nombres serán M1...M14. M1 corresponde al *minterm* 0000, M2 al 0001, y así hasta M14, que corresponde al *minterm* 1111. En cuanto a los implicantes primos, los denominaremos IP1, IP2,... IP*k*. En el ejemplo hay 12 IPs, cuyos nombres serán IP1 el correspondiente a --00; IP2, a --11; y así hasta IP12, que corresponde al último de ellos, 10--.

En primer lugar, fijémonos bien en cuál es el objetivo buscado con el tratamiento de la tabla de cobertura anterior. Se trata de encontrar una combinación de implicantes primos tal que:

1) Cubra la totalidad de *minterms*, es decir, todas las columnas de la tabla, y
2) Además, que sea mínima, es decir, que contenga el menor número de implicantes primos posible.

Vamos a fijarnos en la primera de las condiciones necesarias, que todas las columnas estén cubiertas por la combinación dada. Llamemos *Z* a esta función lógica. ¿Qué se necesita para que se cumpla *Z*, es decir, que su valor de verdad sea uno? Es de Perogrullo: ha de cumplirse que todas y cada una de las columnas M*i* queden cubiertas. Llamando F*i* a la función de cobertura de la columna M*i*, entonces debe ocurrir que todas las funciones F*i* deben cumplirse, es decir, tener un valor de verdad de uno, para que la función global *Z* se cumpla a su vez.

O sea: *Z* = (la columna M1 debe estar cubierta) **Y** (la columna M2 debe estar cubierta) **Y** ...
... **Y** (la columna M*n* debe estar cubierta)

Como hemos llamado F*i* a la función "la columna M*i* debe estar cubierta", entonces la función *Z* es, naturalmente, un producto booleano de todas las funciones F*i* referidas a las columnas de la tabla. Es decir, *Z*=F1*F2*....*F*n*. El valor de verdad de *Z* es uno si y sólo si los valores de verdad de todas las funciones F*i* son uno a su vez, es decir, si todas las columnas están cubiertas.

Fijémonos ahora en cómo son cada una de estas funciones F*i*. Para que la columna *i-ésima* esté cubierta en la combinación dada es preciso que al menos uno de los distintos implicantes primos IP*k* que la cubren esté incluido en la solución, es decir, que al menos uno de los IPs que tienen una X en la intersección con dicha columna esté incluido en la solución. Puede haber en dicha solución más de un IP que cubra la columna , pero basta con que solamente uno de ellos esté incluido en la solución para que la columna *i-ésima* esté cubierta, y eso se representa, claro está, con una suma booleana.

Si la columna *i* es cubierta por *k* implicantes primos distintos, es decir, hay *k* equis en la columna *i-ésima*, para que F*i* se cumpla debe cumplirse que:

(el IP1 esté seleccionado en la solución) **O** (el IP2 esté seleccionado en la solución) **O** ...

... **O** (el IP*k* esté seleccionado en la solución), es decir, la función (IP1+IP2+...IP*k*).

Éste es, por lo tanto, el contenido de la función F*i* antes definida. Sustituyendo estos valores de F*i* en la expresión anterior, *Z*=F1*F2*....F*n*, lo que resulta es **un producto de sumas** que expresa las condiciones necesarias para que se cumpla la condición necesaria de que todas las columnas de la tabla estén cubiertas.

Vale, ya sé que todo esto no es fácilmente comprensible. Recurramos al ejemplo anterior a ver si ayuda a comprender la base de todo esto. Miremos la tabla de cobertura del ejemplo. Para que las 14 columnas de la tabla estén cubiertas es preciso que todas y cada una de ellas lo estén. Comencemos por la primera de ellas, la columna M1 correspondiente al *minterm* 0000. Para que resulte cubierta, la combinación de IPs elegida debe obligatoriamente incluir al menos a uno de los tres IPs que cubren esta columna: el IP1 (--00), el IP4 (-00-) ó el IP8 (0-0-). Son ellos tres los únicos que tienen una X en la intersección con la primera columna.

Por lo tanto, podemos definir esta función F1 como (IP1+IP4+IP8). Análogamente podemos definir la función F2, correspondiente a la columna M2, como (IP3+IP4+IP7+IP8), pues estos son los cuatro IPs que tienen una equis en la columna M2. F3 será (IP2+IP3+IP7), y así hasta llegar a F14, que será (IP2+IP6+IP11). Finalmente todas estas funciones se combinan entre sí multiplicándose entre ellas mediante productos lógicos, dando origen a la larguísima expresión *Z* siguiente:

Z=(IP1+IP4+IP8)*(IP3+IP4+IP7+IP8)*(IP2+IP3+IP7)*(IP1+IP5+IP8+IP9)*
(IP7+IP8+IP9)*(IP5+IP6+IP9)*(IP2+IP6+IP7+IP9)*(IP1+IP4+IP10+IP12)*
(IP3+IP4+IP12)*(IP10+IP11+IP12)*(IP2+IP3+IP11+IP12)*(IP1+IP5+IP10)*
(IP5+IP6+IP10+IP11)*(IP2+IP6+IP11)

Son catorce términos, uno por columna, en este caso concreto con entre 3 y 4 IPs cada uno, dependiendo del número de equis que hay en cada columna.

¿Qué significa esta expresión? Sencillo: la expresión *Z* anterior, derivada de la propia tabla de cobertura de implicantes primos y *minterms*, contiene el compendio de todas las posibles formas de cubrir todas las columnas de la tabla con los IPs de las filas.

Por ejemplo, una posible forma de cubrir todas las columnas sería elegir IP1, porque cubre la columna M1; IP4, porque cubre la columna M2; IP7, la columna 3 ... y por fin IP11, que cubre la columna M14. Por tanto, por ejemplo, la siguiente combinación de IPs:

{IP1, IP4, IP7, IP5, IP8, IP6, IP2, IP1, IP3, IP10, IP3, IP5, IP11, IP11} es una combinación válida que garantiza que todas las columnas están cubiertas. En ella hay IPs repetidos que obviamente deben eliminarse para dejar así la combinación elegida:

{IP1, IP4, IP7, IP5, IP8, IP6, IP2, IP3, IP10, IP11}

Como ésta hay otras muchas combinaciones que cubren todas las columnas: basta con escoger uno cualquiera de los IPs de cada término, de cada suma, para conformar una solución perfectamente válida.

Conocida esta *Z*, que asegura cuando su valor de verdad es uno que todos los *minterms* están cubiertos, ahora hay que seleccionar la combinación de IPs que, haciendo verdadera la expresión *Z*, contenga el menor número posible de implicantes primos. Y esto no es sencillo.

En el caso del ejemplo anterior existen 26.873.856 formas de crear combinaciones válidas de IPs: 8 de las columnas tienen 3 equis, mientras que las otras 6 tienen 4 equis; por lo tanto el número de combinaciones posibles es $3^8 \cdot 4^6 = 6561 \cdot 4096 = 26.873.856$. Aunque ese número se reducirá mucho debido a la existencia de numerosos casos con IPs repetidos, de todos modos los restantes siguen siendo muchos miles. En cualquier caso, encontrar la solución mínima no es obvio, ni mucho menos. ¿Cómo podemos encontrar de forma viable a partir de la expresión *Z* la o las combinaciones válidas que contengan el menor número posible de implicantes primos?

Pues utilizando otro inteligente artificio booleano basado en que, por construcción, la expresión *Z* es un Producto de Sumas, es decir, está en Forma Normal Conjuntiva. Las reglas del álgebra de Boole, en concreto la propiedad distributiva, permiten operar con dicha expresión hasta convertirla en otra equivalente cuya forma sea una Suma de Productos, es decir, ponerla en Forma Normal Disyuntiva. Se trata de una expresión lógica expresada de forma diferente, pero con la misma tabla de verdad, es decir, se trata en realidad la misma expresión lógica que *Z*. En un momento veremos para qué sirve esta transformación.

Recordemos aquí, por un lado, que la Idempotencia indica que a*a=a y que a+a=a; y por otro, que la Ley de Absorción indica que a+a*b=a y que a*(a+b)=a. Ambas leyes booleanas se usarán en las transformaciones que siguen, además de la propiedad distributiva, en concreto (a+b)*(c+d)=a*c+a*d+b*c+b*d.

El método para transformar un Producto de Sumas en una Suma de Productos no puede ser más sencillo. Largo, sí, pero sencillo. Vamos a verlo con un pequeño ejemplo, distinto del utilizado hasta ahora, pero que por su sencillez permite comprender bien el proceso:

Sea *Z*=(IP1+IP2)*(IP2+IP3+IP5)*(IP2+IP4)

Es decir, en la tabla de cobertura hay cinco IPs en filas y tres *minterms* en columnas. En primer lugar aplicamos la propiedad distributiva en los dos primeros términos. Para ello se combinan de dos en dos todos los términos del primer término (IP1+IP2) con cada uno de los tres términos del segundo término (IP2+IP3+IP5):

(IP1+IP2)*(IP2+IP3+IP5) =
(IP1*IP2+IP1*IP3+IP1*IP5+IP2*IP2+IP2*IP3+IP2*IP5)

Aplicando ahora las reglas del álgebra de Boole en la expresión resultante del paso anterior, por Idempotencia queda que IP2*IP2=IP2, y por la Ley de Absorción, IP2+IP2*IP3=IP2, o también IP1*IP2+IP2=IP2, etc. Entonces, operando de esta forma en la expresión anterior, resulta la fórmula siguiente:

(IP1*IP3+IP1*IP5+IP2)

Éste es, pues, el resultado de multiplicar los dos primeros términos de *Z*; ahora, para finalizar la conversión hay que multiplicar este resultado obtenido por el tercer y último término de *Z*, (IP2+IP4).

(IP1*IP3+IP1*IP5+IP2)*(IP2+IP4) =
(IP1*IP3*IP2+IP1*IP3*IP4+IP1*IP5*IP2+IP1*IP5*IP4+IP2*IP2+IP2*IP4), y reduciendo:
(IP1*IP3*IP4+IP1*IP5*IP4+IP2)

Esto es largo y tedioso, pero sencillo: únicamente hemos aplicado las reglas del álgebra de Boole. Por lo tanto, la expresión *Z* original, un Producto de Sumas obtenido en principio a partir de una cierta tabla de cobertura, que era *Z*=(IP1+IP2)*(IP2+IP3+IP5)*(IP2+IP4), es idéntica a la expresión recién encontrada, *Z*=IP1*IP3*IP4+IP1*IP5*IP4+IP2, es decir, ambas tienen la misma tabla de verdad, pero en cambio ahora la expresión tiene la forma de Suma de Productos.

¿Qué significa ahora esa nueva función *Z* expresada como Suma de Productos? Es sencillo: ahora las tornas han cambiado, pues para que la función *Z* se cumpla basta con que se cumpla uno solo de los términos, es decir, uno de los productos que están sumados en la expresión. Con que el valor de verdad de un único término sea uno, el valor de verdad de la función *Z* es uno. Traduciendo esta información a nuestro problema, lo que quiere decir esta función *Z* como Suma de Productos es lo siguiente: **Cada uno de los términos de esa Suma de Productos representa una combinación diferente de implicantes primos, combinaciones que cumplen todas ellas la condición necesaria de cubrir todas las columnas de la tabla, los *minterms***.

En el pequeño ejemplo tratado, la combinación de IPs {IP1, IP3, IP4} que forman el primer término cubre todas las columnas de la tabla y es, por lo tanto, válida. También lo es la combinación {IP1, IP5, IP4} formada por el segundo término, y lo es también la combinación {IP2}, que aunque tenga un solo implicante primo es tan válida como las demás. A partir de aquí, seleccionar la combinación mínima, la que contiene menos IPs, es trivial: es la combinación o combinaciones que tengan un menor número de IPs multiplicados dentro de los productos. En este caso es evidentemente IP2 la solución mínima de este miniejemplo, ejemplo que no tiene mucho sentido en el mundo real dado que el implicante primo IP2 está en todas las sumas, y eso quiere decir que cubre él solito todos los *minterms*... Valga exclusivamente como ejemplo para facilitar la comprensión del proceso, que, repito, no es sencillo de comprender.

Hasta aquí el método de Petrick que, como anticipaba, en realidad lo que hace es convertir una expresión en Forma Normal Conjuntiva (Producto de Sumas) en otra equivalente expresada en Forma Normal Disyuntiva (Suma de Productos).

Hasta ahora he definido este método de Petrick exclusivamente en base a las reglas del álgebra de Boole: a diferencia del resto de Pasos, no he descrito algoritmo informático alguno para hacerlo. Veamos cómo puede implementarse esta conversión de Petrick de forma sencilla.

En primer lugar hay que asignar a cada implicante primo de la tabla un código que tendrá tantos bits como implicantes primos tenga la tabla de cobertura. En nuestro ejemplo de unas páginas más arriba son doce los implicantes primos de la tabla, luego ésa, doce, será precisamente la longitud en bits de los códigos de IP. Estos códigos de IPs se crean de la siguiente manera: el código del IP *i-ésimo* tendrá todos sus bits a cero excepto el bit *i-ésimo*, que será un uno. De ahí que la longitud del código sea el número de IPs de la tabla, pues todos estos códigos tienen un único bit a uno y todos los demás bits a cero.

En el ejemplo, el primer IP de la tabla de cobertura, el --00, tendrá un uno en su primer bit y ceros en el resto, es decir, 100000000000; el segundo, el --11, un uno en su segundo bit y ceros en el resto de bits, o sea, 01000000000, y así hasta el duodécimo y último, el 10--, cuyo código será 000000000001.

Así preparados los códigos, se necesitan para implementar el algoritmo tres tablas de códigos de IPs, que llamaremos Tabla 1, Tabla 2 y Tabla Resultado. Y el algoritmo es:

1) Se inicializa la Tabla 1 con un único elemento ficticio de valor todo ceros.
2) Para cada columna de la tabla de cobertura se ejecuta el proceso siguiente:
   2.1 Se cargan en la Tabla 2 todos los códigos de los IPs de la tabla de cobertura que tengan una X en la columna tratada.
   2.2 Se fusionan dos a dos todos los códigos contenidos en la Tabla 1 con todos los códigos contenidos en la Tabla 2, llevando el resultado a la Tabla Resultado. La fusión entre códigos se hace mediante un OR lógico de los bits de ambos códigos, que son de la misma longitud por construcción. Es decir, si cualquiera de los dos bits en la misma posición es 1, el bit resultante es un 1; si ambos bits son 0, el resultado es 0.
   En este paso, si la Tabla 1 contiene *n* elementos y la Tabla 2 contiene *m* elementos, el número de elementos resultantes en la Tabla Resultado será de $n \cdot m$, dado que cada elemento de la Tabla 1 debe fusionarse con todos los elementos de la Tabla 2, dando origen a un elemento en la Tabla Resultado por cada pareja de elementos.
   2.3 Se procesa la Tabla Resultado para eliminar elementos redundantes, en base a la aplicación de la Ley de Absorción y la Idempotencia.
   Para ello se compara cada elemento de la Tabla Resultado con todos los otros elementos de la propia Tabla Resultado, de la forma:
   Para cada pareja de elementos comparados, se fusionan con un OR lógico ambos elementos en un área de trabajo. Se compara entonces dicha área de trabajo con los dos elementos tratados. Si fuera igual a uno cualquiera de los dos, se elimina dicho elemento igual. Si fuera igual a ambos, se elimina solamente uno de ellos.
   O, visto de otra manera:
   - Si los dos elementos comparados son exactamente iguales, salvo que uno de ellos tiene algunos unos adicionales donde el otro tiene ceros, entonces puede suprimirse el que tiene más unos (se trata de una Absorción).
   - Si los dos elementos comparados tienen exactamente los mismos unos y en las mismas posiciones, entonces puede suprimirse uno cualquiera de ellos (se trata de una Idempotencia), pero solamente uno.

   2.4 Se copian desde la Tabla Resultado todos los códigos restantes, aquellos que no han sido suprimidos, a la Tabla 1, y se continúa con la siguiente columna, hasta haber tratado todas las columnas de la tabla de cobertura.
3) La interpretación del resultado final en la Tabla 1 es la siguiente: cada elemento es un término, un producto de IPs, y todos los elementos de la Tabla 1 están sumados entre sí. Es decir, cada código presente en la Tabla 1 final representa a una cierta combinación de IPs que cubre todas las columnas de la tabla de cobertura. Los IPs que forman dicha combinación son los que corresponden a los unos del código presente en dicha Tabla 1. Cualquiera de los términos, es decir, cada uno de los elementos contenidos en la Tabla 1 al final del proceso genera una solución válida que cubre todas las columnas.
   Por tanto, para localizar la solución mínima basta con revisar la Tabla 1 contando el número de unos de los elementos allí presentes, y quedarse con el que tenga un menor número de unos, o uno cualquiera de ellos si hay varios.
4) Una vez seleccionado el elemento con menor número de unos, se revisan los bits de dicho elemento. Si en la posición *i-ésima* hay un 0, se ignora; si hay un 1, entonces se lleva el implicante primo de la fila *i-ésima* a la Tabla de IPs seleccionados, aquella que se había ido rellenando con los implicantes primos esenciales que se habían encontrado, y a partir de la cual se genera en el Paso 8 la expresión final buscada por el algoritmo.

Una característica a destacar de este algoritmo de Petrick es que en el proceso de fusión y comparación de los elementos de las diferentes tablas se utilizan sentencias OR, que son mucho más rápidas de ejecución que, por ejemplo, la fusión de implicantes primos que se efectúa en el **Paso 3 – Cálculo de los implicantes primos**, dado que los IPs a fusionar en ese Paso 3 pueden tener tres valores diferentes: 0, 1 y guión, lo que obliga a realizar un cierto y no muy eficiente proceso iterativo sobre los valores de los IPs.

Veamos cómo funciona el método de Petrick en el ejemplo que había definido antes, con 12 IPs en filas y 14 *minterms* en columnas.

Repito a continuación la tabla de cobertura del ejemplo:

	***Minterms* de la expresión**													
IPs	**0000**	**0001**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1110**	**1111**
--00	X			X				X				X		
--11			X				X				X			X
-0-1		X	X						X		X			
-00-	X	X						X	X					
-1-0				X		X						X	X	
-11-						X	X						X	X
0--1		X	X		X		X							
0-0-	X	X		X	X									
01--				X	X	X	X							
1--0								X		X		X	X	
1-1-										X	X		X	X
10--								X	X	X	X			

En primer lugar se asignan los códigos a cada implicante primo tal como se indica a continuación. El tamaño de los distintos códigos será en este caso de 12 bits, pues 12 son los IPs de la tabla.

IP	**Código asignado**
--00	100000000000
--11	010000000000
-0-1	001000000000
-00-	000100000000
-1-0	000010000000
-11-	000001000000
0--1	000000100000
0-0-	000000010000
01--	000000001000
1--0	000000000100
1-1-	000000000010
10--	000000000001

Se inicializa la Tabla 1 con un único elemento a ceros, técnica que sirve para que la primera ejecución del bucle de columnas se limite a copiar el contenido de la Tabla 2 a la Tabla 1. Por tanto, en la ejecución correspondiente a la segunda columna el contenido inicial de las Tablas 1 y 2 es el siguiente:

Tabla 1
100000000000
000100000000
000000010000

Tabla 2
001000000000
000100000000
000000100000
000000010000

La Tabla 1 contiene los códigos de los tres IPs que cubren la primera columna, es decir, que tienen una equis en la intersección entre dicho IP y el *minterm* correspondiente a la columna tratada, la primera. Estos IPs son IP1, IP4 e IP8. En cuanto a la Tabla 2, contiene los códigos de los cuatro IPs que cubren la segunda columna, que son IP3, IP4, IP7 e IP8. Comprobad que es así en la tabla de cobertura anterior, si lo deseáis.

Ahora se fusionan todos los códigos de la Tabla 1 con todos los de la Tabla 2. El resultado será una tabla de 12 elementos (3·4), y su contenido es el siguiente:

Tabla Resultado
101000000000
100100000000
100000100000
100000010000
001100000000
000100000000
000100100000
000100010000
001000010000
000100010000
000000110000
000000010000

En efecto, el primer elemento de la Tabla 1, que es 100000000000, al fusionarse con el primero de Tabla 2, 001000000000, recordemos que mediante un OR lógico, da como resultado el elemento 101000000000; con el segundo de Tabla 2, 000100000000, da 100100000000; etc. Luego se fusiona el segundo elemento de la Tabla 1, 000100000000, con los cuatro de la Tabla2, y luego el tercero, 000000010000, también con los cuatro de la Tabla 2, y así hasta la fusión de los dos últimos elementos de cada tabla, que como ambos son idénticos, 000000010000, dan como resultado exactamente el mismo código, 000000010000 (esto es así como consecuencia de la Idempotencia: IP8*IP8=IP8).

Ahora, antes de seguir con la tercera columna hay que reducir en lo posible esta Tabla Resultado, eliminando aquellos elementos cuyos unos coinciden en número y posición con los de otro elemento, salvo que pueden tener un número adicional de unos. Veamos:

Sean los elementos 100100000000 (el segundo de la Tabla) y 000100000000 (el sexto). Todos los unos del sexto de ellos son también unos en las mismas posiciones en el segundo de ellos, que además tiene algún uno más, en concreto el situado en la primera posición. Por lo tanto, el primero de ellos, 100100000000, puede ser eliminado, pues es absorbido por el segundo. Esta situación concreta representa la fórmula IP1*IP4+IP4, que se convierte en IP4, eliminando el término IP1*IP4 debido a la Ley de Absorción. Una forma alternativa de realizar esta comprobación es, ya lo adelanté, fusionando ambos elementos con OR en un área de trabajo, lo que da como resultado 100100000000, que es exactamente igual que el segundo elemento de la tabla, y por tanto se puede eliminar dicho elemento, el segundo, pues es absorbido por el sexto.

Aplicada sistemáticamente esta comprobación a todos los elementos de la Tabla Resultado, se eliminan los elementos: 100100000000, 100000010000, 001100000000, 000100100000, 000100010000, 001000010000, 000100010000 y 000000110000, por lo que la Tabla Resultado queda finalmente con los elementos siguientes:

Tabla Resultado
101000000000
100000100000
000100000000
000000010000

Esta Tabla Resultado donde se han eliminado los elementos absorbidos se copia ahora completa a la Tabla 1, y se continúa el proceso tratando la siguiente columna, en este caso la tercera de la tabla de cobertura, que tiene tres equis, por lo que la Tabla 2 queda como se puede ver a continuación:

Tabla 1
101000000000
100000100000
000100000000
000000010000

Tabla 2
010000000000
001000000000
000000100000

El resultado de realizar la fusión entre ambas tablas en la Tabla Resultado, que nuevamente tendrá doce elementos (4·3, al ser cuatro los elementos en la Tabla 1 y tres los de la Tabla 2), es el siguiente:

Tabla Resultado
111000000000
101000000000
101000100000
110000100000
101000100000
100000100000
010100000000
001100000000
000100100000
010000010000
001000010000
000000110000

Tabla donde, una vez eliminados los elementos redundantes, aquellos que son absorbidos por algún otro elemento (111000000000, 101000100000, 110000100000 y 101000100000), queda finalmente con los 8 elementos siguientes:

Tabla Resultado
101000000000
100000100000
010100000000
001100000000
000100100000
010000010000
001000010000
000000110000

Como la cuarta columna, que es la siguiente a tratar, tiene 4 equis, en la siguiente iteración la Tabla Resultado tendrá 32 elementos (8·4), que quedan reducidos a 12 tras eliminar los redundantes.

El algoritmo sigue procesando el resto de columnas, con cada vez más elementos involucrados en las Tablas, hasta que tras tratar la ultima columna, la número 14, quedan finalmente 58 elementos en la Tabla Resultado... habrá que creerme, o más bien creer al programa que lo calcula, porque no voy yo a detallar aquí todos los cálculos hasta llegar a esa cifra, que el lector armado de paciencia puede realizar si lo desea para comprobar que efectivamente la cifra es correcta.

Es decir, la expresión *Z* inicial, calculada como Producto de Sumas en base a la tabla de cobertura y convertida por obra y gracia del método de Petrick en una Suma de Productos, tiene 58 términos, 58 productos lógicos que se suman entre sí para obtener dicha expresión *Z*. Cada uno de esos 58 términos representa una combinación de implicantes primos que cubren todas las columnas de la tabla de cobertura. Por ejemplo, uno de los términos finales es 010101010100, que representa a la combinación de IPs {IP2, IP4, IP6, IP8, IP10}, combinación que en efecto cubre todos los *minterms*; otro de los términos que hay en la tabla es el 111010001001, que representa a la combinación {IP1, IP2, IP3, IP5, IP9, IP12}, etc.

Ahora basta con encontrar cuál de estos 58 elementos requiere menos IPs, es decir, tiene menos unos, para localizar la solución mínima que cubre todas las columnas, que es el objetivo final del algoritmo QM.

En este ejemplo hay cinco de los 58 términos totales que contienen solamente cuatro unos, y por tanto necesitan de solamente cuatro implicantes primos para cubrir la totalidad de las columnas. Son los siguientes: 101000001010, 100001100001, 010100001100, 010010010001 y 001001010100.

Queda clara aquí una característica inherente al método de Petrick: se trata de un método exhaustivo, encuentra no sólo *una* solución mínima, sino que las encuentra *todas ellas*; en algunos casos puede ser conveniente elegir una u otra en función de ciertos condicionantes. En nuestro caso, donde lo que deseamos es optimizar el código de un programa fuente, elegiremos la combinación que menos variables individuales precise, es decir, que tenga un menor número total de ceros y unos o, si lo preferimos, un mayor número de guiones entre todos los implicantes primos de la solución, pues así se garantiza que el programa fuente resultante tiene el menor número posible de condiciones a evaluar.

En el ejemplo, todos los IPs tienen dos guiones, por lo que a priori da igual una que otra solución, podemos elegir cualquiera de ellas para obtener la solución mínima buscada, por ejemplo el primero de ellos, el 101000001010, que da origen a la combinación {IP1, IP3, IP9, IP11}, y entonces estos cuatro implicantes primos, que son{--00, -0-1, 01--, 1-1-}, se llevan a la Tabla de IPs seleccionados, que en este ejemplo concreto estaba de momento vacía, dado que no se había detectado ningún implicante primo esencial en los pasos anteriores.

Así queda finalmente dicha tabla:

IPs Seleccionados
--00
-0-1
01--
1-1-

Y esta lista de IPs seleccionados da por fin origen, según el Paso 8 del algoritmo de Quine-McClusky explicado anteriormente, a la expresión siguiente: N3*N4+N2*C4+N1*C2+C1*C3. Ordenando la fórmula queda **C1*C3+N1*C2+N2*C4+N3*N4**. No es posible extraer aquí ningún factor común, ni tampoco había ninguna variable implicante, por lo que ésta es, definitivamente, la expresión mínima que tiene como forma canónica a 1101111111111011. Una de ellas, puesto que como hemos visto existen otras cuatro combinaciones más que usan también cuatro IPs, el mínimo absoluto.

Y, no lo olvidemos, ésta es la solución mínima si se ejecuta el algoritmo QM tratando los *minterms*, los unos. Faltaría comprobar qué ocurriría si se tratan los ceros, los *maxterms*, para obtener la que es efectivamente la expresión mínima que tiene la forma canónica dada.

Resumiendo: El método de Petrick es un método muy sencillo de implementar una vez se entienden los procelosos artificios booleanos en que se basa, método que siempre proporciona una expresión mínima cuya forma canónica es la dada; es más, devuelve todas las posibles combinaciones que contienen un número mínimo de variables. De hecho devuelve *todas* las combinaciones de implicantes primos de cualquier tamaño que cumplen la condición de cubrir todas las columnas de la tabla de cobertura.

El método de Petrick sólo tiene un pero, un "pero" ciertamente importante: **corre en tiempo exponencial**, aunque atemperado por la eliminación de términos derivada de la aplicación de la Ley de Absorción o la Idempotencia, que van eliminando ciertos elementos de las tablas, un número apreciable de ellos en muchas ocasiones. Sin embargo, conforme crece el número de implicantes primos o, aún peor, el número de columnas de la tabla de cobertura, el número de elementos necesarios en las tres Tablas definidas crece y crece hasta volverse completamente inmanejable.

Supongamos una sencilla tabla de cobertura con 16 implicantes primos y 20 columnas, y supongamos que cada columna tiene cuatro equis, números habituales, incluso pequeños, en cuanto la forma canónica tiene 32 bits o más. Si no se eliminara ningún elemento de las tablas debido a la Ley de Absorción o a la Idempotencia, el número de productos de que se compondría la Suma de Productos final sería de 4 (número de equis por columna) elevado a 20 (el número de columnas). Y 4^{20} son muchísimos elementos para poder tratarlos, ni tan siquiera poder almacenarlos, pues son más de un billón (europeo), nada menos que 1.099.511.627.776 elementos, concretamente.

Por mucho que los procesos de eliminación de elementos reduzcan sensiblemente el tamaño de las tablas, cosa que efectivamente hacen, éstas crecen y crecen exponencialmente de forma inexorable hasta desbordar todos los buffers plausibles.

Además, hay que tener en cuenta que el propio proceso de comprobación de elementos en la Tabla Resultado para eliminar los elementos redundantes también necesita de un tiempo considerable cuando crece el número de elementos: siendo n el número de elementos de dicha Tabla Resultado tras la fusión de los elementos de las Tablas 1 y 2, es preciso realizar $n \cdot (n-1)/2$ comparaciones de elementos para determinar si es posible eliminar uno u otro, pues deben compararse todas las posibles parejas de códigos. Por tanto, cuando el número de elementos crece, el tiempo de verificación de elementos también crece de acuerdo al cuadrado del número de elementos de la tabla.

Para una Tabla Resultado de por ejemplo 5.000 elementos, cifra que se alcanza con facilidad en cuanto la tabla de cobertura tiene más allá de 25 ó 30 columnas, este paso de comprobación debe comparar casi 12 millones y medio de pares de elementos, es decir, el tiempo requerido por este proceso no es despreciable en absoluto cuando el número de elementos contenidos en las tablas crece por encima de algunos miles.

En definitiva, al igual que ocurre con el método de fuerza bruta, y aunque requiera menos capacidad de cómputo para realizarlo, intentar aplicar el método de Petrick para resolver tablas de cobertura con más de 35, 40 ó quizás 50 columnas se vuelve totalmente impracticable, por muchas eliminaciones de elementos de las tablas que se hagan por el camino.

Para terminar estos párrafos dedicados sobre todo al método de Petrick sólo queda indicar que, como todo en álgebra de Boole es dual, este método puede usarse también, sin cambio alguno, para realizar la transformación contraria, es decir, para convertir una Suma de Productos en un Producto de Sumas. No me extiendo más sobre esta conversión, dado que no es el objeto de este documento sobre minimización de funciones lógicas, pero el lector no hallará dificultad alguna en corroborarlo.

Todas las referencias al método de Petrick que he encontrado en la Red lo hacen como parte del algoritmo de Quine-McClusky. Se cita en la literatura como si su única utilidad fuera resolver la tabla de cobertura de implicantes primos que genera al algoritmo QM, lo que no es poco, pero es que en realidad **el método de Petrick para lo que sirve es para transformar una expresión en Forma Normal Conjuntiva en una equivalente en Forma Normal Disyuntiva, y viceversa,** siempre que sólo haya variables afirmativas, sin complementar, en la expresión. Nada más. Y nada menos.

Para poder utilizarlo para resolver la tabla de cobertura se realizan, como ya he explicado, astutos artificios lógicos asimilando las filas, los IPs, a las variables de la expresión, y las columnas, los *minterms*, a los términos, es decir, las sumas, de la expresión. Una vez realizada esta asimilación, el algoritmo hace su trabajo convirtiendo la expresión en FNC a FND, y un nuevo inteligente artificio lógico permite seleccionar la combinación que menor número de IPs contiene para dar respuesta al Paso 7 del algoritmo de Quine-McClusky.

Leyendo los artículos que hay en la Red sobre el método de Petrick no me queda claro si el método se circunscribe a los artificios booleanos descritos o abarca también el método en sí para transformar una expresión que sea un Producto de Sumas a otra equivalente expresada como Suma de Productos, siempre que todas las variables estén en modo afirmativo. Porque esta transformación se puede utilizar en cualquier problema que lo requiera, no sólo para encontrar la expresión mínima equivalente a una dada.

Aquí se acaba el algoritmo de Quine-McClusky tal como se describe normalmente en la literatura, con un endemoniado penúltimo paso devorador de recursos en cuanto la forma canónica supera un cierto tamaño.

Entonces, si queremos minimizar una expresión lógica de cierto tamaño... ¿se acaba aquí el camino? ¿No hay manera de encontrar en un tiempo razonable la solución mínima, o al menos una lo suficientemente cercana a la mínima correspondiente a una determinada forma canónica cuando ésta crece por encima de un determinado número de bits?

Cuando esto pasa, la consecuencia que generalmente tiene es que el número de implicantes primos y el de columnas de la tabla de cobertura crece más allá del límite a partir del cual el consumo de recursos hace virtualmente imposible calcularla ni por fuerza bruta ni por el método de Petrick; tan sólo queda probar el incierto camino de los algoritmos de programación entera, como el *branch&bound*... por cierto, en el artículo de la Wikipedia inglesa sobre el método de Petrick se afirma que este método de Petrick también es denominado "*branch&bound*"... Se trata de un error, obviamente, pues se trata de dos métodos muy diferentes.

En definitiva, según la documentación que yo he encontrado en la Red del algoritmo de Quine-McClusky, no hay más. No parecen existir más algoritmos o procesos que permitan tratar la tabla de cobertura.

Sin embargo, sí que hay otros caminos, otros métodos, otros algoritmos procedentes de otras áreas de la informática que son aplicables a este problema. Para encontrar estos métodos alternativos hay, en primer lugar, que mirar a la tabla de cobertura de IPs con otros ojos.

Ésta es la tabla de cobertura de implicantes primos y *minterms* que hasta ahora hemos usado como ejemplo, correspondiente a la forma canónica 1101111111111011, con 12 filas (IPs) y 14 columnas (*minterms*):

	***Minterms* de la expresión**													
IPs	**0000**	**0001**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1110**	**1111**
--00	X			X				X				X		
--11			X				X				X			X
-0-1		X	X						X		X			
-00-	X	X						X	X					
-1-0				X		X						X	X	
-11-						X	X						X	X
0--1		X	X		X		X							
0-0-	X	X		X	X									
01--				X	X	X	X							
1--0								X		X		X	X	
1-1-										X	X		X	X
10--								X	X	X	X			

El objetivo del Paso 7 del algoritmo de Quine-McClusky es, como hemos visto, localizar el conjunto de filas (IPs) que, cubriendo la totalidad de las columnas (*minterms*) de la tabla, requieran del menor número posible de IPs (filas) para ello. Pero si nos olvidamos por un momento de la minimización de funciones booleanas y miramos ahora la tabla en términos de **conjuntos**, podemos expresar el mismo problema de la siguiente manera:

"Dado un conjunto de elementos {1, 2, 3...*m*}, al que llamaremos Universo, y *n* conjuntos cuya unión comprende el Universo, se debe identificar el menor número de conjuntos tales que su unión aún contiene a todos los elementos del Universo".

Traduciendo: en nuestro ejemplo tenemos 14 elementos en el Universo, que en este caso lo forman los *minterms*: el 0000, el 0001,... hasta el 1111; y 12 conjuntos, los IPs, que comprenden cada uno de ellos un subconjunto de elementos del Universo, las equis en la intersección con las columnas. El IP --00, por ejemplo, consta de los elementos del Universo {0000, 0100, 1000, 1100}; el IP --11, de los elementos {0011, 0111, 1011, 1111}, y así con todos los IPs. E *identificar el menor número de conjuntos tales que su unión aún contenga a todos los elementos del Universo* es exactamente lo mismo que *localizar la combinación de IPs que hacen mínima la expresión booleana correspondiente a la forma canónica dada*. Solamente hemos cambiado la forma de definir el problema, que sigue siendo el mismo.

Pues bien, este problema así definido es muy conocido: es el "***Set Cover Problem***", es decir, el "**Problema de Cobertura de Conjuntos**", problema que es *NP-Completo* y que forma parte de la lista de 21 problemas *NP-Completos* de Karp.

Este problema ha sido muy estudiado a lo largo de los años, tanto que la investigación sobre él ha dado origen al desarrollo de un conjunto de técnicas que resultan fundamentales para los algoritmos de aproximación.

Entonces ocurre que **los dos problemas citados**, el de búsqueda de la expresión mínima en la tabla de cobertura de IPs propio del algoritmo de Quine-McClusky para la minimización de funciones lógicas, por un lado, y el Problema de Cobertura de Conjuntos, por otro, **son, en realidad, el mismo problema**.

Y por tanto, y esto es importante, **todos los métodos y técnicas utilizados para resolver uno de ellos son directamente aplicables para resolver el otro**.

Esto quiere decir que, por ejemplo, el método de Petrick es perfectamente utilizable para resolver el Problema de Cobertura de Conjuntos, así como lo es el *branch&bound*. Y viceversa, todos los algoritmos y métodos desarrollados para resolver el Problema de Cobertura de Conjuntos son directamente aplicables, sin cambios, al Paso 7 del algoritmo de Quine-McClusky, la búsqueda de la solución mínima.

La constatación de que ambos problemas son exactamente el mismo problema y que, por tanto, cualquier técnica utilizada en uno de ellos es aplicable directamente a la resolución del otro no la he visto reflejada en ningún sitio en la Red, donde ambos problemas, la minimización de expresiones lógicas y la cobertura de conjuntos, viven plácidas vidas separadas. No digo que sea yo el primero en darse cuenta de la igualdad de ambos problemas, mucho me extrañaría tal cosa; lo que digo es que no he encontrado nada en la Red acerca de esta igualdad.

Por ejemplo, el problema de la Cobertura de Conjuntos puede ser resuelto, una vez confeccionada la tabla de cobertura, mediante los Pasos 5 a 7 del algoritmo de Quine-McClusky: selección de IPs esenciales (en este caso serían *conjuntos esenciales*), eliminación de IPs (*conjuntos*) redundantes y aplicación, por ejemplo, del método de Petrick para la búsqueda de la solución mínima.

Y al contrario, todos los algoritmos creados para resolver la cobertura de conjuntos son directamente aplicables para resolver el Paso 7 - Búsqueda de la solución mínima del algoritmo de Quine-McClusky, que es el que nos interesa en este documento.

Veamos cuáles son esos algoritmos específicamente desarrollados para el Problema de Cobertura de Conjuntos y cómo pueden aplicarse a la minimización de expresiones lógicas. Descartando la omnipresente fuerza bruta, son básicamente dos:

- **Programación lineal entera**. Se define una función económica, en este caso el número de implicantes primos utilizados, y el algoritmo trata de minimizar dicha función económica cumpliendo una serie de restricciones, en este caso que todas las columnas, los *minterms*, hayan sido cubiertos.
  El algoritmo se detiene cuando la mejora de la función económica obtenida en un cierto paso es menor que un cierto límite, y por tanto el método no garantiza la obtención del resultado óptimo.
  De hecho, *branch&bound* es un método de programación lineal entera.

- **Algoritmo Voraz (*Greedy Algorithm*).** Es un sencillo algoritmo iterativo, en el que en cada iteración se selecciona el implicante primo que cubre más *minterms* de la tabla de cobertura. El IP seleccionado se elimina de la tabla de cobertura, así como todas las columnas que cubre, exactamente igual que se hacía con los IPs esenciales, y se prosigue así hasta que todas las columnas de la tabla han sido eliminadas.
  Es éste un algoritmo muy sencillo de definir, comprender e implementar y que, según la literatura, es básicamente el mejor algoritmo de aproximación a la solución en tiempo polinómico para resolver el problema de la cobertura de conjuntos en supuestos de "complejidad plausible", sea eso lo que sea. Suponiendo cierta esta afirmación, la consecuencia inmediata es que este algoritmo **tiene que ser también el mejor algoritmo de aproximación para localizar el conjunto mínimo de implicantes primos que cubren la totalidad de *minterms* de la tabla de cobertura**.

Vista esta circunstancia, he programado también este Algoritmo Voraz para resolver el Paso 7 del algoritmo de Quine-McClusky, una vez eliminados en el Paso Conjunto 5-6 todos los IPs esenciales y los redundantes. La primera constatación que surge es que, debido a su sencillez, **es el único método de los propuestos al que no le importa demasiado el número de filas (implicantes primos), ni tampoco el de columnas (*minterms*) para encontrar una solución en un tiempo muy razonable**.

Lo que hay que valorar es con cuanta frecuencia esta solución encontrada es, además, la que contiene el menor número posible de IPs, es decir, es la mínima. Y resulta que... bueno, mejor lo vemos con el ejemplo.

La aplicación del Algoritmo Voraz original tal como lo he descrito más arriba requiere, en cada iteración, seleccionar el implicante primo que más columnas restantes cubra. En caso de que haya varios IPs que cubran el mismo número de columnas se elige uno al azar, por ejemplo el primero de ellos.

Entonces, en nuestro ejemplo se elige en primer lugar el primero de todos los IPs, el --00, que cubre cuatro columnas como todos los demás, puesto que inicialmente todos los IPs de la tabla tienen cuatro equis, es decir, cubren cuatro columnas.

Una vez seleccionado dicho IP --00 se eliminan de la tabla tanto el IP seleccionado como las columnas que este IP cubre, que son los *minterms* 0000, 0100, 1000 y 1100, y tras estas eliminaciones ambas tablas quedan así:

IPs	0001	0011	0101	0110	0111	1001	1010	1011	1110	1111
--11		X			X			X		X
-0-1	X	X				X		X		
-00-	X					X				
-1-0				X					X	
-11-				X	X				X	X
0--1	X	X	X		X					
0-0-	X		X							
01--			X	X	X					
1--0							X		X	
1-1-							X	X	X	X
10--						X	X	X		

IPs seleccionados
--00

En la segunda iteración se selecciona el primer IP que cubra cuatro columnas, pues cuatro sigue siendo el máximo local. Se selecciona entonces el --11, el primero de ellos, y tras las eliminaciones pertinentes queda el siguiente contenido en las tablas:

IPs	0001	0101	0110	1001	1010	1110
-0-1	X			X		
-00-	X			X		
-1-0			X			X
-11-			X			X
0--1	X	X				
0-0-	X	X				
01--		X	X			
1--0					X	X
1-1-					X	X
10--				X	X	

IPs seleccionados
--00
--11

En esta tercera iteración todos los IPs que han quedado cubren solamente dos columnas, así que se selecciona uno cualquiera, nuevamente el primero de todos ellos, el -0-1, y ambas tablas quedan así:

IPs	0101	0110	1010	1110
-00-				
-1-0		X		X
-11-		X		X
0--1	X			
0-0-	X			
01--	X	X		
1--0			X	X
1-1-			X	X
10--			X	

IPs seleccionados
--00
--11
-0-1

Ahora el primer IP que más columnas cubre, que son nuevamente dos, es el –1-0. Entonces se selecciona dicho IP, se eliminan la fila y las dos columnas cubiertas, y queda:

IPs	0101	1010
-00-		
-1-0		
-11-		
0--1	X	
0-0-	X	
01--	X	
1--0		X
1-1-		X
10--		X

IPs seleccionados
--00
--11
-0-1
-1-0

Y en esta tabla restante cada IP restante cubre una única columna, por lo que se puede seleccionar en esta quinta iteración cualquiera de ellos, es decir, el primero, el 0--1, y en la sexta cualquiera de los que cubren la columna restante, en concreto el 1--0, por lo que la tabla final de IPs seleccionados es la siguiente:

IPs seleccionados
--00
--11
-0-1
-1-0
0--1
1--0

La expresión final que encuentra este Algoritmo Voraz es, pues, la siguiente:

N3*N4+C3*C4+N2*C4+C2*N4+N1*C4+C1*N4

Con seis IPs y doce variables, **no es ésta en absoluto la solución óptima**.

Eso ya lo sabemos, puesto que cuando resolvimos este mismo problema mediante el método de Petrick llegamos a una expresión de tan sólo cuatro términos y ocho variables individuales. Claro que esta reflexión tiene sentido mirando la expresión que devuelve el Voraz tal como la entrega al algoritmo, porque si ahora aplicamos a dicha expresión la propiedad distributiva, sacando como factores comunes a las condiciones C4 y a N4, la expresión que queda finalmente es: **N4*(C1+C2+N3)+C4*(N1+N2+C3)**, que, qué casualidad, contiene también ocho variables individuales, las mismas que se obtenían por el método de Petrick.

No obstante, y obviando la posible reducción obtenida al sacar factores comunes, parece claro que el Algoritmo Voraz es susceptible de sufrir cambios que mejoren su desempeño. Encontrar qué cambios mejoran la solución obtenida y cuáles no ha resultado ser un proceso muy divertido para mí, planeando un cambio, programándolo y probándolo para constatar, en la mayoría de ocasiones, que dicho cambio no servía para nada... pero algunos sí que han resultado eficaces para mejorar la solución obtenida por el algoritmo con un coste razonable.

Pero, claro, para saber si un cambio funciona bien o no hay que valorar metódicamente su comportamiento, para poder comparar unos con otros. Y, claro, en primer lugar lo que hay que escudriñar es cómo es el desempeño del Algoritmo Voraz en su versión original. Veamos:

En todas las posibles formas canónicas de 4 variables, es decir, de 16 bits, 65.534 casos en total que se ejecutan dos veces cada una, una vez seleccionando los unos de la forma canónica y otra seleccionando los ceros, es decir, en 131.068 ejecuciones en total, la aplicación del Algoritmo Voraz en su forma original, tal como lo he descrito más arriba, no encuentra la solución mínima en exactamente 3.120 ocasiones, lo que representa un 2,4% de los casos.

Con formas canónicas de más variables, y hasta donde he podido comprobar sobre la base de simulaciones con formas canónicas generadas pseudoaleatoriamente, la proporción aumenta paulatinamente con el número de variables (y el tamaño de la forma canónica): para 5 variables la tasa de error es del 4,5%; para 6 variables, del 10,4%. A partir de ahí sube rápidamente hasta cotas superiores al 22% para formas canónicas de 7 variables. Y por encima de eso seguro que es aún peor, claro, pero es imposible para mí tener datos concretos debido al tiempo necesario para buscar cuál es la expresión mínima absoluta (mediante ataque por fuerza bruta o por el método de Petrick) para formas canónicas de esos tamaños y poder comparar ambas soluciones.

Vistos estos datos, mucho me temo que para formas canónicas de 1.024 o más bits la tasa de acierto será muy baja. En estos casos de formas canónicas de longitudes superiores a 1.024 casi podemos afirmar que el algoritmo Voraz proporciona de forma sencilla y eficiente *una* solución, una aproximada, una expresión que resuelve la forma canónica dada, pero de ahí a afirmar que esta solución sea óptima... sólo de forma testimonial lo será.

En fin, ¿es un 2,4% de error mucho o poco, o un 4,4%, o un 10%, un 25%, etc. según el caso? ¿Es aceptable o no lo es? Se trata de un algoritmo muy sencillo de implementar y rápido de ejecutar que para formas canónicas de 16 bits tiene una tasa de acierto del 97,6%, lo que a priori suena bien, pero que baja rápidamente hasta un 75% y menos, mucho menos, conforme aumenta el tamaño de la forma canónica.

Supongo que si es aceptable o no esa cifra depende de la aplicación que se vaya a dar al resultado de la fórmula. Para mí, perfeccionista irredento, incluso ese nimio 2,4% de error es mucho. Así que, como ya he dicho, he diseñado, programado y probado diversas adiciones de código al algoritmo para intentar mejorar su tasa de acierto. Unas iban bien y las conservaba; otras, rematadamente mal y las tiraba a la papelera. Describo a continuación el algoritmo resultante, aquel con el que mejor resultado he conseguido para todos los tamaños de forma canónica que he probado.

Esta modificación al algoritmo original se basa en que en muchas ocasiones, y en cualquier iteración del algoritmo, existen varios implicantes primos que cubren el mismo número máximo de *minterms*. Por ejemplo, en la segunda iteración del ejemplo teníamos cinco IPs que cubrían el máximo local, que eran cuatro columnas; el resto de IPs cubrían menos columnas, dos o tres. Cuando se da esta circunstancia, el algoritmo original selecciona uno cualquiera de ellos, normalmente el primero. Pues bien, la idea es que en estos casos, en lugar de seleccionar a voleo un IP cualquiera de los que cubren el número máximo de columnas, se trata de revisar cómo son cada uno de estos IPs, y seleccionar no uno cualquiera, sino el mejor de todos ellos. Pero ¿cuál de ellos es aquí *el mejor*?

Tras muchas pruebas, he llegado al siguiente proceso para determinarlo:

1) Todos los implicantes primos que cubren la cifra máxima de columnas, y solamente ellos, se almacenan en una tabla *ad hoc* para tratarlos. Tabla que, como máximo, tendrá la misma dimensión que la tabla de cobertura.
2) Se calculan para cada uno de los IPs contenidos en esta tabla *ad hoc* dos cifras, que son: **a)** el número de equis que, como máximo, quedarían en cualquier otro IP de la propia tabla *ad hoc* en caso de que se seleccionara finalmente el implicante primo tratado; y **b)** el número total de equis que quedarían en el resto de IPs de la tabla *ad hoc* si se seleccionara el IP tratado.
3) Tras el proceso anterior se selecciona finalmente el IP que más equis deje en cualquier otro IP de la tabla *ad hoc* (cifra calculada en el paso **2.a**) anterior).
4) A igualdad de esta cifra entre varios IPs, se selecciona el que más equis deje en el conjunto de IPs de la tabla *ad hoc* (cifra calculada en el paso **2.b**) anterior).
5) A igualdad entre varios IPs en ambas cifras, se selecciona el que más guiones contenga en su código.
6) Si tras todas estas comparaciones persiste la igualdad entre varios IPS se elige uno cualquiera de ellos, el primero, por ejemplo.

Se trata, de alguna manera, de tomar el IP que una vez seleccionado, y por tanto eliminados de la tabla tanto el propio IP como las columnas que cubre, menos impacto tenga en el resto de IPs que cubren el mismo número de columnas, maximizando el número de columnas que estos otros IPs de la tabla *ad hoc* cubren tras su selección.

En otras palabras, se busca que los conjuntos de equis cubiertos por el IP seleccionado y por el resto de IPs sean lo más disjuntos posible, pero sólo entre los IPs que cubren el número máximo de columnas. Si se ejecuta este mismo proceso para todos los IPs de la tabla de cobertura, entonces la solución encontrada finalmente es casi siempre sensiblemente peor que si se circunscribe el proceso a los pocos IPs que cubren dicho número máximo de columnas. Ignoro la razón, pero así es lo que he comprobado empíricamente.

Este algoritmo Voraz modificado mejoró sustancialmente la tasa de acierto en las formas canónicas tratadas. Veamos los datos:

Para formas canónicas de 16 bits y cuatro variables se llega a una tasa de acierto del 99,62% (sólo en 496 casos de los 131.068 posibles no halló la combinación mínima; recordad que el proceso se ejecuta dos veces para cada forma canónica, una tratando los *minterms*, los unos, y otra, los ceros, los *maxterms*). Para 32 bits y 5 variables se llega a una tasa de acierto del 98,61% de los casos; para 64 bits y 6 variables, se acierta en el 95,81% de los casos, y por fin, para 128 bits y 7 variables la tasa de acierto mejoró hasta el 90,49% de los casos. Más allá de estos tamaños no he realizado más comprobaciones por las razones que ya he explicado.

En definitiva, se trata de una modificación que mejora muy significativamente las tasas de acierto, puesto que, como puede verse, dicha tasa de acierto más que duplica la del Algoritmo Voraz en su forma original.

Veamos ahora qué resultados da este Algoritmo Voraz modificado que acabo de definir a partir de la tabla de cobertura del ejemplo que estamos siguiendo.

Recordemos una vez más cómo es esa tabla de cobertura del ejemplo:

	Minterms de la expresión													
IPs	**0000**	**0001**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**	**1010**	**1011**	**1100**	**1110**	**1111**
--00	X			X				X				X		
--11			X				X				X			X
-0-1		X	X						X		X			
-00-	X	X						X	X					
-1-0				X		X						X	X	
-11-						X	X						X	X
0--1		X	X		X		X							
0-0-	X	X		X	X									
01--				X	X	X	X							
1--0								X		X		X	X	
1-1-										X	X		X	X
10--								X	X	X	X			

En la primera iteración todos los IPs cubren cuatro columnas, luego todos ellos se copian a la tabla *ad hoc* definida. Allí se calculan el número de equis que como máximo quedarían en cualquier otro IP de la propia tabla *ad hoc*, que son siempre cuatro en este caso, pues en cualquier caso quedarán varios IPs que cubren cuatro columnas, y también el número total de equis que quedarían en el total de IPs restantes, que en este caso, debido a la configuración especial de la tabla al no haber IPs esenciales, son siempre 34. Contadlos, si no lo veis claro.

Como además todos los IPs tienen dos guiones, son todos igual de elegibles, así que se selecciona el primero de ellos, el --00. La tabla de cobertura queda así (de momento, igual que con el algoritmo sin modificar, al ser seleccionado también en este caso el primer IP, el --00):

IPs	**0001**	**0011**	**0101**	**0110**	**0111**	**1001**	**1010**	**1011**	**1110**	**1111**
--11		X			X			X		X
-0-1	X	X				X		X		
-00-	X					X				
-1-0				X					X	
-11-				X	X				X	X
0--1	X	X	X		X					
0-0-	X		X							
01--			X	X	X					
1--0							X		X	
1-1-							X	X	X	X
10--						X	X	X		

En la tabla de IPs seleccionados está solamente el --00 recién seleccionado, dado que no había ningún IP esencial.

IPs seleccionados
--00

En la segunda iteración el número máximo de equis cubiertas sigue siendo cuatro, pero ahora sólo los IPs --11, -0-1, -11-, 0--1 y 1-1-, cinco en total, cubren cuatro columnas. Se llevan todos ellos a la nueva tabla *ad hoc*, y se calcula, exclusivamente entre ellos, cuántas equis quedarían en caso de ser seleccionados, en un solo IP y en el conjunto de ellos.

Esa tabla *ad hoc* queda así una vez cargada:

	Tabla *ad hoc*		
IP	**Num.X otro IP**	**Num.X totales**	**Num.guiones**
--11	2	8	2
-0-1	4	11	2
-11-	4	11	2
0--1	4	11	2
1-1-	4	11	2

Por ejemplo, en el IP --11, de seleccionarse este IP, es decir, si se eliminaran las cuatro columnas que cubre, resulta que los otros cuatro IPs de esta tabla *ad hoc* sólo cubrirían dos columnas, y entre todos ellos sumarían, por lo tanto, ocho equis a cubrir. Sin embargo, con el IP -0-1, de seleccionarse, y una vez eliminadas las columnas cubiertas por él, vemos que el IP -11- seguiría cubriendo cuatro columnas, es decir, los conjuntos de columnas cubiertas de estos IPs, el -0-1 y el -11-, son disjuntos. Si contamos el número total de equis que restarían en el total de los otros cuatro IPS de seleccionarse el -0-1, son: 4 del -11-, 2 del --11, 2 del 0--1 y 3 del 1-1-, en total, por lo tanto, 11.

Así se calculan los datos para todos los IPs de la tabla *ad hoc* y se obtienen los resultados que se muestran más arriba. Es un cálculo que no requiere excesivo tiempo para ejecutarse, pues la tabla *ad hoc* tiene normalmente muchos menos elementos que la propia tabla de cobertura.

Como los cuatro últimos IPs de esta tabla *ad hoc* tienen los mismos datos, sólo se descarta el --11, que tiene menores números de equis restantes, por lo que se selecciona el primero del resto, que en este caso es el -0-1, y la tabla de cobertura resultante queda así:

IPs	**0101**	**0110**	**0111**	**1010**	**1110**	**1111**
--11			X			X
-00-						
-1-0		X			X	
-11-		X	X		X	X
0--1	X		X			
0-0-	X					
01--	X	X	X			
1--0				X	X	
1-1-				X	X	X
10--				X		

IPs seleccionados
--00
-0-1

En la tercera iteración el número máximo de equis cubiertas sigue siendo cuatro, pero ahora sólo el IP -11- cubre esas cuatro columnas. No hay mucho más que pensar, se selecciona dicho IP y la tabla de cobertura queda ahora así:

IPs	**0101**	**1010**
--11		
-00-		
-1-0		
0--1	X	
0-0-	X	
01--	X	
1--0		X
1-1-		X
10--		X

IPs seleccionados
--00
-0-1
-11-

En la cuarta iteración tampoco hay mucho que hacer, los IPs que todavía cubren alguna columna solamente lo hacen con una, y todos sus datos son idénticos en la tabla *ad hoc*, una equis restante en cualquier otro IP, y tres restantes en total, por lo que se selecciona el primero de ellos, el 0--1, y en la quinta y última iteración quedan sólo los tres IPs que cubren la última columna a tratar, la del *minterm* 1010. En fin, todos tienen los mismos datos en la tabla *ad hoc*, por lo que se selecciona el primero de ellos, el 1--0.

Como consecuencia, la tabla de IPs seleccionados queda definitivamente así, con cinco IPs, que ya se ve que son uno menos que los seis que conformaban la solución encontrada aplicando el Algoritmo Voraz sin cambio alguno:

IPs seleccionados
--00
-0-1
-11-
0--1
1--0

La expresión obtenida es ahora la siguiente:

N3*N4+N2*C4+C2*C3+N1*C4+C1*N4, que también puede reducirse sacando factores comunes a N4 y a C4: **N4*(N3+C1)+C4*(N2+N1)+C2*C3**.

Aplicando esta modificación al Algoritmo Voraz, modificación que realmente no implica un gran consumo adicional de recursos, se mejora de forma importante el desempeño del algoritmo para encontrar la solución óptima, como ya comenté: prácticamente se más que duplica la tasa de acierto sobre el algoritmo en su estado original.

Pero sigue habiendo un porcentaje de fallos, aunque menor que antes: un 0,38% en formas canónicas de 16 bits; un 1,39% en 32 bits; un 4,19% en 64 bits; un 9,51% en 128 bits, etc. Sigue siendo un porcentaje apreciable en cuanto la expresión tenga más de cinco variables, es decir, cuando la forma canónica tiene 64 bits o más... De hecho, con este Algoritmo Voraz modificado hemos llegado en la forma canónica del ejemplo a una expresión de cinco términos y diez variables individuales, pero sabemos que se puede mejorar y conseguir una solución de cuatro términos y solamente ocho variables individuales, puesto que esta misma expresión la hemos tratado con el método de Petrick y ya sabemos cuál es aquí el resultado óptimo. Por lo tanto, ésta del ejemplo es precisamente una de esas 496 formas canónicas de 16 bits para las que este Algoritmo Voraz modificado no encuentra la solución mínima absoluta.

¿Se puede mejorar aún más este Algoritmo Voraz para que su índice de acierto se aproxime al 100%? Sí, se puede, aunque a costa de aumentar significativamente, ahora sí, el tiempo necesario para su ejecución, aunque manteniéndose siempre en tiempo polinómico.

Investigando algunos de los casos en los que este algoritmo modificado no encuentra la solución óptima, verbigracia el del ejemplo, se trata de casos en los que cierta combinación de IPs que no son los que más columnas cubren en cada iteración se complementan entre sí de tal modo que cubren eficazmente la totalidad de columnas. Entonces, la solución pasa por obligar al algoritmo a seleccionar esos IPs que no se seleccionarían siguiendo el devenir normal del método, pero que son necesarios para conformar la solución óptima. Para ello se añade un bucle previo al Algoritmo Voraz modificado como lo he descrito antes, en el que se selecciona obligatoriamente un IP de los existentes en la tabla de cobertura en cada iteración, y se ejecuta a continuación el algoritmo modificado anteriormente descrito con el resto de IPs.

Es decir, el Algoritmo Voraz modificado descrito en los párrafos anteriores se ejecuta ahora tantas veces como IPs tenga la tabla de cobertura, seleccionando de antemano en cada iteración un IP diferente, y al final de la iteración nos quedamos con la solución que, de todas las obtenidas, menos términos o variables requiera.

El tiempo de ejecución se incrementa de forma importante, pues siendo n el número de IPs en la tabla y T el tiempo necesario para ejecutar el algoritmo modificado, ahora se requiere un tiempo de $n \cdot T$, aunque este T resulte ser algo menor porque el algoritmo se ejecuta ahora con un implicante primo menos, el que ha sido previamente seleccionado en el bucle externo.

Los resultados de las pruebas realizadas son espectaculares. Ahora, en formas canónicas de 16 bits (4 variables) sólo falla en 64 casos de los 131.068 posibles (son 65.534 casos que se ejecutan dos veces cada uno, una tratando los unos, y otra, los ceros), lo que da una tasa de acierto del 99,95%. Para formas canónicas de 32 bits ha fallado en 343 de 400.000 casos probados, un 99,91% de acierto. En formas de 64 bits, un 99,62% de acierto; de 128 bits, un 98,78% de acierto...

No he probado este algoritmo para formas canónicas de 256 bits o más generadas aleatoriamente porque en la práctica totalidad de estos casos el número de IPs y columnas en la tabla de cobertura excede el límite que me permite aplicar el método de Petrick o la fuerza bruta para comprobar la tasa real de acierto. Sólo puedo extrapolar los resultados para formas canónicas de mayor tamaño, y deben tener una tasa de acierto cercana al 95%-98%, lo que, en mi opinión, garantiza una solución óptima en un elevado porcentaje de casos.

Pero... resulta que la forma canónica del ejemplo es, por una *extraña casualidad*, una de esas 64 formas canónicas de 16 bits que este algoritmo modificado no resuelve bien.

Si deseamos seguir aumentando la eficacia del algoritmo, entonces se puede modificar nuevamente el algoritmo anterior para que en cada iteración del bucle se seleccionen obligatoriamente no uno, sino *dos* implicantes primos... Esto tiene consecuencias dramáticas en el tiempo requerido para completarlo: para seleccionar cada par posible de IPs entre los n IPs de la tabla de cobertura es preciso realizar ahora $n \cdot (n-1)/2$ ejecuciones del Algoritmo Voraz modificado (combinaciones de n elementos tomados de 2 en 2), en vez de las n que requería antes. Sí, el algoritmo sigue corriendo en tiempo polinómico, pero en cuanto el número de IPs de la tabla crece por encima de un cierto número se vuelve inmanejable, aunque es cierto que los límites aquí son sensiblemente superiores a los existentes en caso de atacar el problema por fuerza bruta (20-25 IPs) o por el método de Petrick (40-50 columnas).

En efecto, si la tabla de cobertura tuviera 100 IPs debería ejecutarse el Algoritmo Voraz modificado tal como lo acabo de describir 4.950 veces, lo que, dado el hecho de que el propio Algoritmo Voraz modificado no es excesivamente costoso en recursos, puede ser perfectamente asumible. Incluso para 250 IPs, una cifra ya muy respetable, la ejecución de este algoritmo modificado 31.125 veces es aún plausible si se desea afinar al máximo el resultado. Desde luego, pensar en usar el método de Petrick o el de fuerza bruta en tablas de cobertura de ese tamaño es completamente imposible.

La pregunta que cualquiera se plantearía aquí es: ¿cuánto mejora la eficacia del Algoritmo Voraz esta nueva modificación, que es seleccionar previamente todas las posibles parejas de IPs antes de ejecutar el Algoritmo Voraz modificado, y quedarse con la solución mínima entre todas ellas?

Juzgad por vosotros mismos:

Tamaño de la forma canónica	Nº Variables	Tasa de acierto
8 bits	3	100%
16 bits	4	100%
32 bits	5	99,9985%
64 bits	6	99.988%
128 bits	7	99,936%
256 bits	8	99,881%

No, no he comprobado formas canónicas de mayor tamaño. Aunque el algoritmo sí que llega a una solución en tiempo razonable, lo que no es factible para mí y mi equipamiento es encontrar la solución mínima absoluta en tiempo razonable para esos tamaños de forma canónica generada aleatoriamente para poder compararla con la obtenida por este algoritmo, ni por fuerza bruta ni por Petrick.

Y sí, los datos son correctos, no me he equivocado al escribirlos: usando esta versión modificada del Algoritmo Voraz, **la tasa de acierto detectado con formas canónicas de hasta 256 bits es siempre superior al 99,88%**. Y, desde luego, todo lo explicado hasta aquí, incluido este algoritmo Voraz modificado descrito, es directamente aplicable al problema para el que fue originalmente concebido: el **Problema de Cobertura de Conjuntos**, mejorando notablemente su desempeño a cambio, eso sí, de utilizar más recursos informáticos para su resolución.

Obviamente, ahora sí que se llega a encontrar la solución óptima en el ejemplo que hemos seguido: si la forma canónica tiene 16 bits, como es el caso, la tasa de acierto es del 100%. De hecho, por si tenéis curiosidad, la expresión obtenida es: N3*N4+N2*C4+N1*C2+C1*C3, correspondiente a los IPs {IP1, IP3, IP9, IP11} que es, claro está, una de las encontradas por el método de Petrick. Para obtenerla ha tenido que seleccionar obligatoriamente a la pareja de IPs {IP1, IP9} (--00 y 01--), que eliminan entre los dos siete columnas de la tabla, pues ambos comparten una equis en la cuarta columna, la correspondiente al *minterm* 0100. Entonces el Algoritmo Voraz modificado ejecutado sobre la tabla restante de diez IPs y siete columnas selecciona primero IP3 (-0-1), que todavía cubre cuatro columnas, y luego IP11 (1-1-), que cubre finalmente las tres columnas restantes.

Bien, dadas estas elevadas tasas de acierto de este Algoritmo Voraz modificado y la relativa simplicidad del propio algoritmo, mi opinión es que hay que pensarse bien si merece la pena implementar para resolver este Paso 7 del algoritmo de Quine-McClusky un método que obtenga siempre la solución mínima (fuerza bruta, Petrick) para conseguir mejoras en la obtención de la solución de quizás un 1 ó un 3 por mil, quizás de un 1%.

Otra opción factible es implementar dos métodos, por ejemplo el de Petrick y este Voraz modificado, pues ninguno de ellos es complicado de programar, y ejecutar uno u otro en función del tamaño de la tabla de cobertura. Y, recordad, lo que siempre se debe hacer es **ejecutar el algoritmo dos veces** con cada forma canónica, **una tratando los unos (*minterms*) y otra tratando los ceros (*maxterms*) y quedarse con la solución que resulte de menor tamaño**.

Como ya demostré en el tercer capítulo de este documento, hay una alta probabilidad, cercana al 50% en cuanto la forma canónica tiene un cierto tamaño, de que la solución obtenida por uno de los dos métodos no sea la mínima absoluta.

Y, además, para encontrar la expresión mínima absoluta correspondiente a la forma canónica dada se debe **aplicar la propiedad distributiva sobre las expresiones obtenidas** para reducir su tamaño extrayendo todos los factores o sumandos comunes que sea posible.

Veamos por curiosidad qué ocurre si para resolver esta rebelde forma canónica del ejemplo tratamos los ceros, los *maxterms*, en lugar de los unos, los *minterms*. Resulta que en la tabla de implicantes primos sólo se cargarían inicialmente dos, que son los correspondientes a los dos ceros de la forma canónica, es decir: 0010 y 1101.

Como 0010 tiene un único uno y 1101 tiene tres, no hay comparaciones posibles entre ellos en el Paso 3, ni, por tanto, fusión alguna de elementos, por lo que estos dos son también los que se cargarían en la tabla de cobertura; cada uno de ellos cubre exclusivamente al *minterm* homónimo, luego ambos son implicantes primos esenciales y se seleccionan, dejando la tabla de cobertura vacía para el Paso 7 del algoritmo, que por consiguiente no se ejecuta.

No hay nada más que hacer, sólo generar la fórmula final: N1*N2*C3*N4+C1*C2*N3*C4, y a continuacíón obtener su complementaria aplicando las Leyes de De Morgan:

(C1+C2+N3+C4)*(N1+N2+C3+N4)

No es posible reducirla más mediante la propiedad distributiva, pues no tiene ningún sumando común. Y qué casualidad, finalmente tiene ocho variables individuales, las mismas que las expresiones obtenidas tanto por el método de Petrick como por el Algoritmo Voraz modificado tratando los unos, pero para llegar a ella estaréis de acuerdo en que hemos necesitado de bastante menos esfuerzo. Ésta del ejemplo es, pues, una de las 8.262 formas canónicas de 16 bits que, como vimos en el capítulo correspondiente, obtienen resultados de la misma longitud, tanto tratando los ceros como tratando los unos, en este caso ocho condiciones.

La Implementación del Paso 7 - Algunas cifras

En este capítulo hemos debatido sobre qué alternativas existen para implementar el **Paso 7 - Búsqueda de la Solución Óptima** dentro de algoritmo de Quine-McClusky. Los métodos más importantes que hemos visto que pueden utilizarse son la fuerza bruta, la programación lineal entera, sobre todo el *branch&bound*, el método de Petrick y el Algoritmo Voraz (Greedy Algorithm). Cada cual tiene sus ventajas e inconvenientes, por lo que a la hora de escribir un programa cuya función sea minimizar funciones lógicas hay que tener en cuenta las fortalezas y debilidades de cada uno.

A la hora de implementar uno u otro es conveniente conocer los límites efectivos para unos u otros, aquellos que, en función del equipamiento informático disponible, permiten resolver el problema en un tiempo razonable. No es lo mismo disponer de un PC por potente que éste sea, con ocho núcleos y 16Gb de memoria como es mi caso, que disponer de un supercomputador para ejecutar allí el algoritmo.

En base a los datos que he obtenido y a las innumerables pruebas que he realizado, la implementación que yo he hecho de este Paso 7 en mis programas es finalmente la siguiente:

- Si no quedan columnas a tratar en la tabla de cobertura no se ejecuta nada. Es decir, si el paso conjunto 5-6 ha cubierto eficazmente todos los *minterms* con los IPs esenciales, entonces obviamente no hay que ejecutar este Paso 7.
- Si la tabla de cobertura tiene en este punto más de 60 implicantes primos o más de 40 columnas (*minterms*), se ejecuta el Algoritmo Voraz modificado tal como he definido antes.
- En otro caso, es decir, con 60 IPs o menos y 40 columnas o menos, se ejecuta el método de Petrick, pero con una salvaguarda: si en cualquier momento las Tablas internas sobrepasan su tamaño máximo, fijado en mi caso en 5.000 elementos, entonces se aborta la ejecución del método de Petrick y se pasa a ejecutar el Algoritmo Voraz modificado.

¿Cómo se comporta esta implementación? Pues depende mucho del tamaño de la forma canónica. He ejecutado el algoritmo muchos miles o millones de veces para cada tamaño de forma canónica de entre 8 y 1024 bits, es decir, de entre 3 y 10 condiciones individuales, la totalidad de formas canónicas posibles para tamaños de 8 ó 16 bits y simulaciones pseudoaleatorias para el resto, y las cifras de desempeño son las siguientes (se representan en la tabla tanto las cifras absolutas como el porcentaje sobre el total de casos):

Tamaño de la F.C.	Total F.Cs. tratadas	No quedan *minterms*	Método de Petrick	Algoritmo Voraz	M. Petrick cancelado	Alg.Voraz directo
8	508	500 **(98,4%)**	8 **(1,6%)**	0	0	0
16	131.068	125.868 **(96,0%)**	5.200 **(4,0%)**	0	0	0
32	2.000.000	1.834.576 **(91,7%)**	165.365 **(8,3%)**	59 **(0,03%)**	59 **(0,03%)**	0
64	1.000.000	847.115 **(84,7%)**	144.072 **(14,4%)**	8.813 **(0,9%)**	8.468 **(0,85%)**	345 **(0,05%)**
128	600.000	411.098 **(68,5%)**	126.286 **(21,1%)**	62.616 **(10,4%)**	36.041 **(6,0%)**	26.575 **(4,4%)**
256	400.000	145.043 **(36,3%)**	74.940 **(18,7%)**	180.017 **(45,0%)**	34.762 **(8,7%)**	145.255 **(36,3%)**
512	200.000	12.369 **(6,2%)**	11.341 **(5,7%)**	176.290 **(88,1%)**	5.942 **(3,0%)**	170.348 **(85,1%)**
1024	100.000	201 **(0,2%)**	63 **(0,1%)**	99.736 **(99,7%)**	25 **(0,02%)**	99.711 **(99,7%)**

El significado de las columnas es:

- Tamaño de la forma canónica: en bits.
- Total F.Cs tratadas: Número total de ejecuciones para ese tamaño de forma canónica.
- No quedan *minterms*: Número de casos en que tras la ejecución de los Pasos 5 y 6 no quedan columnas en la tabla de cobertura (*minterms*), pues todas ellas han sido cubiertas por IPs esenciales, sean estos primarios o secundarios.
- Método de Petrick: Número de casos que han sido resueltos mediante el método de Petrick.
- Algoritmo Voraz: Número de casos que han sido resueltos mediante el Algoritmo Voraz (Greedy Algorithm) modificado. En todos estos casos se ha podido llegar a ejecutar dicho algoritmo por dos caminos: bien porque se ha tenido que cancelar el método de Petrick al haberse superado el tamaño máximo de las tablas intermedias, bien porque el tamaño de la tabla de cobertura excedía el máximo permitido para ejecutar dicho método de Petrick.
- M. Petrick cancelado: Número de casos en los que se ha comenzado a ejecutar el método de Petrick, pero han sido cancelados al superarse el tamaño máximo de las tablas intermedias (5.000 elementos) y se han resuelto definitivamente con el Algoritmo Voraz.
- Alg.Voraz directo: Número de casos en los que la tabla de cobertura excedía el tamaño máximo permitido (60 IPs, 40 *minterms* en mi caso) y como consecuencia se han resuelto directamente con el Algoritmo Voraz.

El dato del número de casos en los que no queda columna alguna en la tabla de cobertura al llegar al Paso 7 del algoritmo es un dato absoluto, es decir, no depende de los límites impuestos para la ejecución del método de Petrick o el Algoritmo Voraz, es un dato intrínseco del algoritmo, dado que es el resultado de la ejecución de los Pasos 5 y 6, que han cubierto todas las columnas con IPs esenciales, primarios o secundarios.

Los datos indican que cuando la forma canónica tiene un tamaño de 128 bits o menos, entonces una amplia mayoría de los casos son completamente resueltos por los pasos 5 – 6: el 98%, el 96% ... el 68%.

A partir de ahí los porcentajes bajan ya al 36% para formas canónicas de 256 bits, al 6% si son de 512 bits, y a partir de ahí el número de casos en que no es necesario ejecutar el Paso 7 al quedar cubiertas todas las columnas es marginal; para formas canónicas de más de 1.024 bits es seguro que dicho porcentaje se acercará mucho a cero.

En cuanto al resto de datos, dependen de los límites impuestos para ejecutar el método de Petrick; si estos fueran más altos de los que tengo marcados (60 IPs, 40 columnas, tablas intermedias de 5.000 elementos), el número de casos en los que dicho método acabaría resolviendo satisfactoriamente la forma canónica ciertamente aumentarían, aunque a costa, eso sí, de un importante incremento en el consumo de recursos para completar el método de Petrick que, recordemos, corre en tiempo exponencial.

El lector puede tomar sus propias decisiones en base a toda esta información.

Hasta aquí llega este documento dedicado a desmenuzar el algoritmo de Quine-McClusky, quizás a mejorarlo y, siempre, a compartir con los lectores todo lo que he aprendido al respecto. Seguro que todo lo que aquí digo es susceptible de mejora; yo he llegado como buenamente he podido hasta donde he llegado, y seguro que se puede ir mucho más allá... pero no yo.

Naturalmente, queridos lectores, podéis criticar lo que aquí cuento, modificarlo o incluso diseñar vuestro propio método y mejorar lo conocido. Lo único que os pido en tal caso es que nos hagáis partícipes de vuestros hallazgos.

Vale.

Macluskey

Enero de 2023