Minimización de Funciones Lógicas
El algoritmo de Quine – McClusky
explicado y mejorado

(I)

Macluskey, 2023

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.

Entonces, ¿por qué dedico los cinco artículos que componen esta miniserie, de los que éste es el primero, 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, por sorprendente que parezca, 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 estos artículos para que quien lo desee los use si le son de utilidad.

Pero antes de entrar en harina, 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; es 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.

El **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 cierta 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.

Por razones que no vienen al caso llevo algunos años trabajando en la optimización de programas fuente, y por lo tanto, 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 una cierta longitud.

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 estos artículos 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, 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 en la serie son resultados reales de diversas ejecuciones, millones de ellas, que he realizado. Y aseguro, además, que los programas que he usado no tienen errores de programación. Mejor dicho, 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 en los distintos artículos de la serie 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 variables afirmativas, mientras que N1, N2, N3, etc. son las negativas: 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, por ejemplo, C2: "*Año = 2022*"; C3: "*Mes > 8*"; C4: "*Código = 1*"; y por fin C5: "*Cancelado = NO*", la expresión anterior sería:

"*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 $1{,}8 \cdot 10^{19}$, 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 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 artículo, el segundo de la serie, 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 segundo artículo 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 artículo de la serie 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, los dos últimos artículos de la serie, el cuarto y el quinto, los 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 los artículos que siguen a éste sea de utilidad para alguno de vosotros, lectores de ElCedazo.

Sois libres de utilizarlo como os plazca.

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