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

(V)

Macluskey, 2023

Tal como decía al final del artículo anterior de esta serie sobre el algoritmo de Quine-McClusky, hay más métodos posibles para encontrar la expresión mínima a partir de la tabla de cobertura de implicantes primos aparte de los métodos citados usualmente en la literatura sobre el algoritmo: el de fuerza bruta, el *branch&bound* y el método de Petrick.

Para encontrar estos métodos alternativos hay, en primer lugar, que mirar a la tabla de cobertura de IPs con otros ojos. Vamos ya con esa otra mirada.

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 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 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***" o "**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 y la investigación sobre él ha dado origen al desarrollo de un conjunto de técnicas 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 que vimos en el artículo anterior 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 óptima.

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 esta serie de artículos. 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 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 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 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, como consecuencia **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 los IPs esenciales y los redundantes. 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 implica, en cada paso, 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 los IPs, el --00, que cubre cuatro columnas como todos los demás (inicialmente todos los IPs tienen cuatro equis), y una vez seleccionado se eliminan de la tabla tanto el IP seleccionado como las columnas que cubre, y queda:

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 lo siguiente 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

Ahora todos los IPs que han quedado cubren sólo dos columnas, así que en el tercer paso se selecciona uno cualquiera, el primero de todos, 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, 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

Ahora cada IP restante cubre una única columna, por lo que se puede seleccionar en esta quinta iteración cualquiera de ellos, y en la sexta, cualquiera de los que cubren la columna restante, 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 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 en el artículo anterior de la serie 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 C4 y a N4, la expresión que queda 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. En primer lugar hay que valorar 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 aleatoriamente, la proporción aumenta paulatinamente con el número de variables: 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 casi 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, pueden existir varios implicantes primos que cubren el mismo número máximo de *minterms*. Por ejemplo, en la segunda iteración del ejemplo había 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 ellos se trata de revisar cómo son cada uno de estos IPs, y seleccionar no uno cualquiera, sino el mejor. Pero ¿cuál 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 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 seleccionar el IP que una vez seleccionado, y por tanto eliminado 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 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.

En definitiva, una mejora muy significativa, como puede verse, que más que duplica la del Algoritmo Voraz en su forma original.

Veamos ahora cómo funciona este Algoritmo Voraz modificado a partir de la tabla de cobertura del ejemplo que estamos siguiendo.

Recordemos antes cómo es esa tabla de cobertura:

	***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 seleccionarse 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 (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 ahora así:

	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 así obtenida es ahora la siguiente:

N3*N4+N2*C4+C2*C3+N1*C4+C1*N4, que también puede reducirse sacando factores comunes: **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, la forma canónica es de 64 bits o más... De hecho, en la forma canónica del ejemplo, con este Algoritmo Voraz modificado hemos llegado 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 las 496 formas canónicas de 16 bits para las que este Algoritmo Voraz modificado no encuentra la solución mínima.

¿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 algoritmo, 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 anterior 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 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* IPs... 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, 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 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, la ejecución del 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, 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 informático 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, 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 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, naturalmente, 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, puesto que 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 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 dije en el tercer artículo de la serie, 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 tercer artículo de la serie, obtienen resultados de la misma longitud, ocho variables en este caso, tanto tratando los ceros como tratando los unos.

La Implementación del Paso 7 - Algunas cifras

En este artículo y en el anterior hemos debatido sobre las alternativas para implementar el **Paso 7 - Búsqueda de la Solución Óptima**. 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, 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 del tamaño de la forma canónica. He ejecutado el algoritmo muchos miles 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 (14,41%)	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.
- 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 que por lo tanto 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 resulta de la ejecución de los Pasos 5 y 6.

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 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 estar 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), el número de casos en los que dicho método acabaría resolviendo satisfactoriamente la forma canónica ciertamente aumentarían, aunque a costa 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.

En fin, hasta aquí llega esta serie dedicada 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