Simplificador de Funciones Booleanas

Proyecto

Universidad Tecnológica Nacional

Facultad Regional Córdoba

Autores:

* Pavan Matías Leg.:
* Federico Prado Leg.:65221

Año 2013

Algoritmo de Quine – McCluskey

El algoritmo de Quine-McCluskey es un método de minimización lógica de dos niveles. Toma una función booleana y trata de reducirla a una fórmula booleana equivalente, más sucinta. Por ejemplo, supongamos que A, B y C son variables booleanas, y la negación lógica de X se escribe como \bar{X}. Entonces podríamos reducir esta fórmula booleana f:

f(A,B,C) = \bar{A}\bar{B}\bar{C} + \bar{A}B + AB\bar{C} + AC

A esta formula equivalente :

f(A,B,C) = B + AC + \bar{A}\bar{C}

Este problema de minimización de fórmulas booleanas también se resuelve mediante mapas de Karnaugh, pero de una manera menos sistemática y escalable.

Representacion de la formula

Vamos a representar a cada término de una fórmula booleana mediante un simple arreglo de bytes. La posición i del arreglo corresponderá a la variable i, y cada una puede tener uno de tres valores:

* **Uno (1),** lo que indica la variable contiene;
* **Cero (0),** lo que indica la negación de la variable mantiene;
* **DontCare**, lo que indica que la variable no está presente en el término y así puede asumir cualquiera de los valores.

Representamos a una fórmula como un simple ArrayList de los arreglos de sus términos. Por ejemplo, podríamos construir la fórmula del ejemplo anterior utilizando este código:

ArrayList<**byte[]**> formula = **new** ArrayList<**byte[]**>**()**;

**byte[]** t1 = **{**0, 0, 0 **}**;

**byte[]** t2 = **{**0, 1, DontCare**}**;

**byte[]** t3 = **{**1, 1, 0 **}**;

**byte[]** t4 = **{**1, DontCare, 1 **}**;

formula.add**(**t1**)**;

formula.add**(**t2**)**;

formula.add**(**t3**)**;

formula.add**(**t4**)**;

Creamos las clases  Term y  Formula para envolver este estado y definir la constante DontCare.

**<<****[Term.java](http://web.archive.org/web/20120210071521/http:/en.literateprograms.org/Quine-McCluskey_algorithm_%28Java%29" \l "chunk use:Term.java)>>=**

**import java.util.\*;**

Declaraciones de “import” de Term

**class** Term **{**

**public** **static** **final** **byte** DontCare = 2;

**public** Term**(byte[]** varVals**)** **{**

**this**.varVals = varVals;

**}**

**public** **int** getNumVars**()** **{**

**return** varVals.length;

**}**

Métodos públicos de Term

**private** **byte[]** varVals;

**}**

**<<****[Formula.java](http://web.archive.org/web/20120210071521/http:/en.literateprograms.org/Quine-McCluskey_algorithm_%28Java%29" \l "chunk use:Formula.java)>>=**

**import java.util.\*;**

Declaraciones de “import” de Formula

**class** Formula **{**

**public** Formula**(**List<Term> termList**)** **{**

**this**.termList = termList;

**}**

Métodos públicos de Formula

Metodos de ayuda de Formula

**private** List<Term> termList;

información adicional de miembros de Formula

**}**

Para comodidad en la depuración (debugging), también añadimos funciones simples para convertir cada uno de estos tipos a cadenas:

**<<****Metodos públicos de Term>>=**

**public** String toString**()** **{**

String result = "{";

**for(int** i=0; i<varVals.length; i++**)** **{**

**if** **(**varVals**[**i**]** == DontCare**)**

result += "X";

**else**

result += varVals**[**i**]**;

result += " ";

**}**

result += "}";

**return** result;

**}**

**<<Metodos públicos de Formula>>=**

**public** String toString**()** **{**

String result = "";

result += termList.size**()** + " terms, " + termList.get**(**0**)**.getNumVars**()** + " variables\n";

**for(int** i=0; i<termList.size**()**; i++**)** **{**

result += termList.get**(**i**)** + "\n";

**}**

**return** result;

**}**

Regla de Resolución

El algoritmo de Quine-McCluskey se basa fundamentalmente en la regla de resolución de la lógica proposicional

En la representación de bits, se puede utilizar esta fórmula para combinar dos términos si:

* Ellos son idénticos excepto por una posición, y
* Esa posición es 0 en un termino y 1 en el otro.

Por ejemplo:

**{**0, 0, DontCare, 1**}**, **{**0, 1, DontCare, 1**}** -> **{**0, DontCare, DontCare, 1**}**

Podemos escribir un método para comprobar las condiciones y realizar la combinación dados dos términos:

**<<Metodos públicos de Term>>=**

**public** Term combine**(**Term term**)** **{**

**int** diffVarNum = -1; // La posición en la que difieren

**for(int** i=0; i<varVals.length; i++**)** **{**

**if** **(this**.varVals**[**i**]** != term.varVals**[**i**])** **{**

**if** **(**diffVarNum == -1**)** **{**

diffVarNum = i;

**}** **else** **{**

// Son diferentes en al menos 2 lugares

**return** **null**;

**}**

**}**

**}**

**if** **(**diffVarNum == -1**)** **{**

// They're identical

**return** **null**;

**}**

**byte[]** resultVars = varVals.clone**()**;

resultVars**[**diffVarNum**]** = DontCare;

**return** **new** Term**(**resultVars**)**;

**}**

A un alto nivel, Quine-McClusky tiene tres pasos básicos:

1. Aplicar esta regla muchas veces para producir términos más pequeños hasta que ya no se puede producir.
2. Identificar los términos que no se pueden combinar con cualquier otro término para producir un término menor. Los llamamos los implicantes primos.
3. Selecciona un subconjunto de estos implicantes primos que implica todos los términos originales.

Encontrando los implicantes primos

Una implementación ingenua examinaría cada par de términos, aplicando la norma siempre que se pueda hasta que se lleve a cabo todas las reducciones posibles.

Podemos reducir drásticamente la búsqueda notando dos hechos:

* Cualquiera de los dos términos que cumplan las condiciones deben tener el mismo número de términos DontCare, ya que la única diferencia que se permite involucra 0 y 1
* Dados dos términos que cumplan las condiciones, uno de ellos debe tener exactamente un 1 más que el otro.

Para tomar ventaja de esto, reorganizamos los datos en una tabla de dos dimensiones de listas. Esta tabla tiene la propiedad de que cada termino que se encuentre en la tabla en el índice table[i][j] tiene i DontCares y j 1 bits. La organizamos de esta manera:

**<<****crear tabla de terminos>>=**

**int** numVars = termList.get**(**0**)**.getNumVars**()**;

ArrayList<Term>**[][]** table = **new** ArrayList**[**numVars + 1**][**numVars + 1**]**;

**for(int** dontKnows=0; dontKnows <= numVars; dontKnows++**)** **{**

**for(int** ones=0; ones <= numVars; ones++**)** **{**

table**[**dontKnows**][**ones**]** = **new** ArrayList<Term>**()**;

**}**

**}**

**for(int** i=0; i<termList.size**()**; i++**)** **{**

**int** dontCares = termList.get**(**i**)**.countValues**(**Term.DontCare**)**;

**int** ones = termList.get**(**i**)**.countValues**((byte)**1**)**;

table**[**dontCares**][**ones**]**.add**(**termList.get**(**i**))**;

**}**

Añadimos uno a las dimensiones de la matriz de modo que los índices pueden asumir todos los valores entre cero y numVars, inclusive. El método countValues ​​de la clase Term simplemente determina el número de variables tienen el valor especificado:

**<<métodos públicos de Term>>=**

**public** **int** countValues**(byte** value**)** **{**

**int** result = 0;

**for(int** i=0; i<varVals.length; i++**)** **{**

**if** **(**varVals**[**i**]** == value**)** **{**

result++;

**}**

**}**

**return** result;

**}**

Ahora, sólo tenemos que tratar de combinar los términos que caen en las listas que se encuentran uno al lado del otro en la tabla. Todavía tenemos que probar todas las combinaciones de estas dos listas, pero con suerte será corto. Si nos encontramos con un par de términos que se pueden combinar, sabemos dónde el resultado va a ir en la tabla: tendrá un DontKnow más que los términos iniciales y un número de unos igual al menor numero de unos de los términos iniciales. Esto es fundamental, no sólo para evitar el conteo innecesario, sino también porque nos dice que podemos hacer una sola exploración de arriba hacia abajo y encontrar todas las combinaciones posibles: ninguna entrada en la tabla que hayamos visitado se modificará.

**<<generar nuevos términos con combine() mientras se actualiza la lista de implicantes primos>>=**

**for(int** dontKnows=0; dontKnows <= numVars - 1; dontKnows++**)** **{**

**for(int** ones=0; ones <= numVars - 1; ones++**)** **{**

ArrayList<Term> left = table**[**dontKnows**][**ones**]**;

ArrayList<Term> right = table**[**dontKnows**][**ones + 1**]**;

ArrayList<Term> out = table**[**dontKnows+1**][**ones**]**;

**for(int** leftIdx = 0; leftIdx < left.size**()**; leftIdx++**)** **{**

**for(int** rightIdx = 0; rightIdx < right.size**()**; rightIdx++**)** **{**

Term combined = left.get**(**leftIdx**)**.combine**(**right.get**(**rightIdx**))**;

**if** **(**combined != **null)** **{**

**if** **(**!out.contains**(**combined**))** **{**

out.add**(**combined**)**;

**}**

Actualizar lista de implicantes primos

**}**

**}**

**}**

**}**

**}**

Una vez que hemos encontrado dos términos que se pueden combinar, sabemos que ninguno de ellos es un implicante primo. Tenemos cuenta de esto, a medida que avanzamos eliminando los dos términos iniciales de termList y añadiendo su combinación:

**<<****actualizar lista de implicantes primos>>=**

termList.remove**(**left.get**(**leftIdx**))**;

termList.remove**(**right.get**(**rightIdx**))**;

**if** **(**!termList.contains**(**combined**))** **{**

termList.add**(**combined**)**;

**}**

Cuando hayamos terminado, la lista contendrá sólo implicantes primos.

Para que la llamada al metodo contains()anterior funcione correctamente, debemos asegurarnos de que equals()tiene la semántica correcta en  Term:

**<<métodos públicos de Term>>=**

**public** **boolean** equals**(**Object o**)** **{**

**if** **(**o == **this)** **{**

**return** **true**;

**}** **else** **if** **(**o == **null** || !getClass**()**.equals**(**o.getClass**()))** **{**

**return** **false**;

**}** **else** **{**

Term rhs = **(**Term**)**o;

**return** Arrays.equals**(this**.varVals, rhs.varVals**)**;

**}**

**}**

Siempre que redifinamos el método equals() tenemos que redefinir el método hashCode() de manera acorde:

**<<[Term public methods](http://web.archive.org/web/20120210071521/http:/en.literateprograms.org/Quine-McCluskey_algorithm_%28Java%29" \l "chunk use:Term public methods)>>=**

**public** **int** hashCode**()** **{**

**return** varVals.hashCode**()**;

**}**

Ponemos la funcionalidad discutida en esta sección en un método de Formula, a continuación. Como vamos a necesitar los términos originales de la siguiente sección, primero los guardamos.

**<<****miembros de datos adicionales de Formula>>=**

**private** List<Term> originalTermList;

**<<****[Formula public methods](http://web.archive.org/web/20120210071521/http:/en.literateprograms.org/Quine-McCluskey_algorithm_%28Java%29" \l "chunk use:Formula public methods)>>=**

@SuppressWarnings**(**"unchecked"**)**

**public** **void** reduceToPrimeImplicants**()** **{**

originalTermList = **new** ArrayList<Term>**(**termList**)**;

[create term table](http://web.archive.org/web/20120210071521/http:/en.literateprograms.org/Quine-McCluskey_algorithm_%28Java%29" \l "chunk def:create term table)

[generate new terms with combine() while updating prime implicant list](http://web.archive.org/web/20120210071521/http:/en.literateprograms.org/Quine-McCluskey_algorithm_%28Java%29" \l "chunk def:generate new terms with combine() while updating prime implicant list)

**}**

Encontrando un conjunto que implique a todos

Nuestro próximo objetivo es encontrar un subconjunto de los implicantes primos que impliquen a todos los términos de la fórmula original. El primer paso es probar exhaustivamente cuáles de los términos originales están implícitos en cuál de los implicantes primos. Para determinar si un término A implica otro término B, sólo tenemos que asegurar que todas las variables a 0 o 1 en A sean las mismas B. Por ejemplo:  {DontCare, 1, 0, DontCare} implica {0, 1, 0, DontCare}. Implementamos esta prueba en un nuevo método en Term:

**<<métodos públicos de Term>>=**

**boolean** implies**(**Term term**)** **{**

**for(int** i=0; i<varVals.length; i++**)** **{**

**if** **(this**.varVals**[**i**]** != DontCare &&

**this**.varVals**[**i**]** != term.varVals**[**i**])** **{**

**return** **false**;

**}**

**}**

**return** **true**;

**}**

Utilizamos esto para construir una nueva tabla, un arreglo bidimensional de booleans donde table[i][j] es verdadero solo si el número implicante *i* implica al termino original numero  *j.*

**<<crear tabla de implicantes >>=**

**int** numPrimeImplicants = termList.size**()**;

**int** numOriginalTerms = originalTermList.size**()**;

**boolean[][]** table = **new** **boolean[**numPrimeImplicants**][**numOriginalTerms**]**;

**for** **(int** impl=0; impl < numPrimeImplicants; impl++**)** **{**

**for** **(int** term=0; term < numOriginalTerms; term++**)** **{**

table**[**impl**][**term**]** = termList.get**(**impl**)**.implies**(**originalTermList.get**(**term**))**;

**}**

**}**

En este punto, estamos frente a una instancia del complejo problema de encubrimiento -set cover problem- “NP”. Las unicas soluciones completas conocidas a este problema son exponenciales, pero también hay estrategias heurísticas simples y eficaces que sean más eficientes.

Si en algún momento un término original en particular está implicado por un solo implicante, decimos que es un implicante primo esencial y debemos usarlo. Quitamos tanto la fila correspondiente a la implicante primo esencial y las columnas de todos los términos originales que implica, ya que estos son cubiertos. Seguimos haciendo esto hasta que cada término restante está implicado por varios implicantes primos.

**<<métodos de ayuda de Formula>>=**

**private** **int** extractEssentialImplicant**(boolean[][]** table**)** **{**

**for** **(int** term=0; term < table**[**0**]**.length; term++**)** **{**

**int** lastImplFound = -1;

**for** **(int** impl=0; impl < table.length; impl++**)** **{**

**if** **(**table**[**impl**][**term**])** **{**

**if** **(**lastImplFound == -1**)** **{**

lastImplFound = impl;

**}** **else** **{**

// This term has multiple implications

lastImplFound = -1;

**break**;

**}**

**}**

**}**

**if** **(**lastImplFound != -1**)** **{**

extractImplicant**(**table, lastImplFound**)**;

**return** lastImplFound;

**}**

**}**

**return** -1;

**}**

El método extractImplicant se encarga de la puesta a cero de la fila y la columna asociada con el implicante dado. Sólo se necesita a cero las columnas, ya que este se encarga de la fila también (cada columna que contiene un “true” en esa fila se borra).

**<< métodos de ayuda de Formula >>=**

**private** **void** extractImplicant**(boolean[][]** table, **int** impl**)** **{**

**for** **(int** term=0; term < table**[**0**]**.length; term++**)** **{**

**if** **(**table**[**impl**][**term**])** **{**

**for** **(int** impl2=0; impl2 < table.length; impl2++**)** **{**

table**[**impl2**][**term**]** = **false**;

**}**

**}**

**}**

**}**

Por último , cubrimos los términos originales restantes utilizando los implicantes primos remanentes. Aquí hay diferentes estrategias posibles a seguir:

El retroceso de búsqueda : en cada paso intentamos todas las opciones posibles , utilizando la regla de "esencial implicante primo " para reducir donde sea posible. Una vez que se encuentre una solución retrocedemos hasta el último punto de decisión con opciones restantes e intentamos la próxima elección. Esta es una solución lenta , exponencial de tiempo, pero siempre dará una fórmula de tamaño mínimo ( óptimo ). Después de encontrar todas las soluciones posibles, elegimos el más pequeño . Nosotros no aplicamos actualmente esta solución.

Selección heurística : Siempre que se enfrenta con una decisión, elegimos el implicante primo que implica el mayor número de términos originales remanentes. Una vez más , se utiliza la regla de "esencial implicante primo " para reducir donde sea posible. Se puede demostrar que la solución resultante es a lo sumo ln n veces más grande que la solución óptima (mínima ), donde n es el número más grande de los términos originales que implica cualquier implicante primo .

**<<métodos de ayuda de Formula>>=**

**private** **int** extractLargestImplicant**(boolean[][]** table**)** **{**

**int** maxNumTerms = 0;

**int** maxNumTermsImpl = -1;

**for** **(int** impl=0; impl < table.length; impl++**)** **{**

**int** numTerms = 0;

**for** **(int** term=0; term < table**[**0**]**.length; term++**)** **{**

**if** **(**table**[**impl**][**term**])** **{**

numTerms++;

**}**

**}**

**if** **(**numTerms > maxNumTerms**)** **{**

maxNumTerms = numTerms;

maxNumTermsImpl = impl;

**}**

**}**

**if** **(**maxNumTermsImpl != -1**)** **{**

extractImplicant**(**table, maxNumTermsImpl**)**;

**return** maxNumTermsImpl;

**}**

**return** -1;

**}**

**<<extraer implicantes de forma heurística hasta acabar >>=**

ArrayList<Term> newTermList = **new** ArrayList<Term>**()**;

**boolean** done = **false**;

**int** impl;

**while** **(**!done**)** **{**

impl = extractEssentialImplicant**(**table**)**;

**if** **(**impl != -1**)** **{**

newTermList.add**(**termList.get**(**impl**))**;

**}** **else** **{**

impl = extractLargestImplicant**(**table**)**;

**if** **(**impl != -1**)** **{**

newTermList.add**(**termList.get**(**impl**))**;

**}** **else** **{**

done = **true**;

**}**

**}**

**}**

termList = newTermList;

También es posible tener soluciones intermedias que combinan heurística con búsqueda limitada - pero una solución que es a la vez de tiempo óptimo y polinomico no se puede lograr a menos que P = NP.

Ponemos esta funcionalidad en otro método en la Fórmula, que contendrá una fórmula reducida cuando regrese:

**<<métodos de ayuda de Formula>>=**

**public** **void** reducePrimeImplicantsToSubset**()** **{**

[create implies table](http://web.archive.org/web/20120210071521/http:/en.literateprograms.org/Quine-McCluskey_algorithm_%28Java%29" \l "chunk def:create implies table)

[extract implicants heuristically until done](http://web.archive.org/web/20120210071521/http:/en.literateprograms.org/Quine-McCluskey_algorithm_%28Java%29" \l "chunk def:extract implicants heuristically until done)

originalTermList = **null**;

**}**