Un algoritmo para la función Safe Contraction

Directora: Dra. Verónica Becher
Autores: Fernando Calderón, Diego Calderón y Norma Marinaro.
Departamento de Computación, FCEyN
Universidad de Buenos Aires.

Diciembre 2001.

Abstract

Esta tesis presenta un algoritmo y una implementación de la función Safe Contraction de Alchourrón y Makinson para teorías de la lógica proposicional. Permite realizar las operaciones de revisión, contracción y expansión.

Contents

1 Introducción

Este trabajo se enmarca en el problema de la representación de los cambios en Inteligencia Artificial. El enfoque clásico de este problema es el de las funciones de cambio de teorías sobre un lenguaje lógico de Alchourrón, Gärdenfors y Makinson (AGM). Estas funciones toman una teoría y una fórmula y dan una teoría actualizada. Aunque el estudio de estas funciones ha recibido gran atención, tanto en su presentación axiomática como en sus posibles construcciones, el problema de dar algoritmos interesantes para las construcciones no ha sido explorado.

En este trabajo presentamos un algoritmo para una de las construcciones más elegantes de las funciones AGM: la función Safe Contraction de Alchourrón y Makinson. Nuestro algoritmo computa la función para teorías de la lógica proposicional basándose en varias heurísticas que lo distancian dramáticamente de la solución por fuerza bruta.

La principal dificultad de trabajar con funciones de cambio sobre teorías es la enorme cantidad de fórmulas que deben ser consideradas. En contraste, computar funciones de cambio sobre bases (conjuntos de fórmulas que no están clausurados por una operación de consecuencia lógica) es más sencillo, y un algoritmo para hacerlo se presenta en la tesis de Christian Durr [11] .

La función de Safe Contraction requiere que se busquen todas las demostraciones que existan dentro de la teoría K para una determinada fórmula por la que se quiere operar. Para ello un algoritmo ingenuo podría generar y evaluar todos los subconjuntos de la teoría a ser actualizada en búsqueda de las demostraciones. Cada subconjunto es un elemento del conjunto de partes de la teoría ($P(K)$) por lo que el algoritmo mencionado necesitaría generar todas los elementos de $P(K)$. Si K tiene m elementos $P(K)$ tiene 2^m elementos.

Puesto que existen infinitas fórmulas lógicamente equivalentes pero sintácticamente distintas, se estaría ante un problema intratable. Para evitar esta situación y dado que no se observa ningún beneficio al tratar explícitamente todas las fórmulas sintácticamente posibles, decidimos utilizar en este trabajo clases de equivalencia. Las fórmulas de la teoría se convierten a una forma normal para trabajar con un único representante de cada clase de equivalencia.

Para n variables proposicionales existen 2^{2^n} fórmulas lógicamente distintas. Una fórmula puede tomar solo dos valores posibles para cada valuación: Verdadero o Falso. Dado que existen 2^n valuaciones, la tabla de verdad de una fórmula tiene 2^n posiciones, luego calculando todas las permutaciones de verdadero y falso en una tabla de verdad obtenemos 2^{2^n} fórmulas. En el siguiente cuadro se muestra la relación entre la cantidad de elementos de K y la cantidad de elementos de $P(K)$ trabajando con 3 variables proposicionales:

K	$\#K$	$\#P(K)$
$Cn(\{a, -a\})$	256	2^{256}
$Cn(\{a, b, c\})$	128	2^{128}
$Cn(\{a, b\})$	64	2^{64}

Cuadro 1

En el cuadro anterior se aprecia la enorme explosión combinatoria que se produce en el conjunto de $P(K)$, por lo tanto para un algoritmo que trabajara ingenuamente el problema se volvería intratable.

La implementación de nuestro algoritmo permite visualizar a las funciones de cambio AGM en su forma más pura y es de interés didáctico. Al momento de realizar esta tesis no sabemos de la existencia de ningún algoritmo similar al planteado ni de ninguna implementación sobre teorías. Existe una implementación sobre bases que fue realizada por Mary-Anne Williams. Información al respecto se puede ver en [10].

Nuestra implementación fue realizada en Borland Delphi 5.0 y puede bajarse de la web en www.dc.uba.ar/tesis/ccm.html. La elección del lenguaje fue basada en su practicidad pero podría haberse elegido cualquier otro lenguaje de propósito general.

El trabajo original donde se presentan las funciones AGM es [2] y la función Safe Contraction es [1]. Para una introducción a las funciones de cambio de teorías y su exposición completa ver [5] y [7].

2 Definiciones básicas

Siguiendo los requerimientos básicos de la teoría AGM asumiremos un lenguaje proposicional $\mathcal{L}$ con los conectivos usuales $(\neg, \vee, \wedge, \rightarrow)$. Denotaremos con $\mathcal{L}$ al conjunto de todas las fórmulas bién formadas, con letras latinas minúsculas a, b, c a las variables proposicionales; f, g, x, y, z a las sentencias, con letras latinas mayúsculas A, B, C, D a los conjuntos de sentencias y reservaremos K para denotar teorías. Consideraremos una relación de consecuencia lógica Cn con las siguientes propiedades:

Sea A un subconjunto de $\mathcal{L}$

i. $A \subseteq Cn(A)$ (inclusion).

ii. Si $A \subseteq B$ entonces $Cn(A) \subseteq Cn(B)$ (monotony).

iii. $Cn(A) = Cn(Cn(A))$ (iteration).

Asumiremos también que la consecuencia lógica incluye la función de implicación tautológica clásica, y las propiedades de deducción y compacidad.

iv. Si x puede ser deducida de A por una función de implicación tautológica clásica entonces $x \in Cn(A)$. (supraclassicality).

v. $x \in Cn(A \cup y)$ si y sólo si $(y \rightarrow x) \in Cn(A)$. (deduction).

vi. Si $x \in Cn(A)$, entonces $x \in Cn(A')$ para algún subconjunto finito $A' \subseteq A$. (compactness).

Una teoría es un conjunto $A \subseteq \mathcal{L}$ tal que $A = Cn(A)$ (cerrado bajo consecuencia lógica). También asumiremos que Cn satisface la regla de "introducción de disyunción en las premisas".

vii. Si $x \in Cn(A \cup \{x_1\})$ y $x \in Cn(A \cup \{x_2\})$ entonces $x \in Cn(A \cup \{x_1 \vee x_2\})$.

Un conjunto $A \subseteq \mathcal{L}$ es consistente sii $Cn(A) \neq \mathcal{L}$ y un conjunto $A \subseteq \mathcal{L}$ es una teoría sii $A = Cn(A)$.

Un conjunto $M \subseteq \mathcal{L}$ es una extensión maximal consistente de $A \subseteq \mathcal{L}$ sii $A \subseteq M$, $Cn(M) \neq \mathcal{L}$ y $\forall x \notin M : Cn(\{x\} \cup M) = \mathcal{L}$.

2.1 Funciones de Cambio AGM

Las tres operaciones definidas por el modelo AGM son *expansión*, *contracción* y *revisión*. La *expansión* es la más simple y consiste en agregar una fórmula a una teoría y cerrar por consecuencia lógica el conjunto expandido.

La expansión se define como:

$$K + x = Cn(K \cup \{x\})$$

La operación de *contracción* consiste en quitar una fórmula de una teoría sin agregar ninguna otra. Al sacar una fórmula x de una teoría K, puede darse el caso que otras fórmulas, o conjuntos de fórmulas de K impliquen x, por lo que no es suficiente quitar sólo x. El problema es determinar qué fórmulas sacar. El modelo AGM proporciona 8 postulados que caracterizan completamente a las funciones de contracción. $- : K \times \mathcal{L} \rightarrow K$

Las funciones de revisión AGM toman una teoría K y una fórmula x y retornan la teoría revisada $K * x$. El problema en la revisión es que x debe ser agregada a K pero el resultado de la operación debe ser una teoría consistente, siempre que x sea satisfactible.

Las funciones de contracción y revisión son mutuamente definibles. Mediante la *identidad de Levi* la revisión puede ser definida en términos de contracción y expansión. Esta identidad define la revisión primero quitando cualquier potencial inconsistencia y luego agregando la nueva fórmula.

$$(\mathbf{Levi - id}) \quad K * x = (K - \neg x) + x.$$

La contraparte de la identidad de Levi es la *identidad de Harper* , que define la contracción en términos de revisión. Las fórmulas de $K - x$ son el resulado de las fórmulas en común entre K y $K * \neg x$.

$$(\mathbf{Harper - id}) \quad K - x = K \cap (K * \neg x).$$

En este trabajo nos concentramos en la contracción y tomamos la revisión como la operación definida a partir de la contracción.

2.1.1 Construcción de las Funciones AGM

Nos concentraremos ahora en el método de construcción de Safe Contraction, que es una de las funciones más elegantes del modelo AGM.

Alchourrón y Makinson [1] dan una construcción de funciones de contracción basadas en órdenes sobre fórmulas del lenguaje $\mathcal{L}$, las funciones $Safe$. Sea A un conjunto de fórmulas y $\prec$ una jerarquía sobre A y sea x una proposición que queremos eliminar de las consecuencias de A, diremos que un elemento es $safe$ respecto de x (módulo $\prec$) si y sólo si no es un elemento mínimo (respecto a $\prec$) de ningún subconjunto minimal B de A tal que $x \in Cn(B)$. Notaremos A/x al conjunto de todos los elementos de A que son $safe$ respecto de x.

La operación de *safe contraction* sobre A (módulo $\prec$) está definida por la ecuación $A - x = A \cap Cn(A/x)$. Si A es una teoría, como $A/x \subseteq A$ tenemos que $Cn(A/x) \subseteq Cn(A)$ por lo tanto:

$$A - x = Cn(A/x).$$

Alchourrón y Makinson [3] y Hans Rott [8] consiguen el teorema de representación donde demuestran que una función es una contracción AGM si y sólo si puede ser construída como una *safe contraction function*, donde $\prec$ es una jerarquía que satisface:

$\prec$ es un orden transitivo no circular

virtually connected : $\prec$ es *virtually connected* sobre A sii para todo x, y, z, si $x \prec y$ entonces $x \prec z$ o $z \prec y$.

continuing up : Para todo $x, y, z \in A$ si $x \prec y$ y $y \vdash z$ entonces $x \prec z$

continuing down : Para todo $x, y, z \in K$ si $x \vdash y$ y $y \prec z$ entonces $x \prec z$

Usaremos fuertemente que:

Sea A un conjunto de fórmulas, $\prec$ una jerarquía sobre A y $B \subseteq A$. Entonces si $\prec$ es *virtually connected* sobre B, $\prec$ cumple *continuing up* sobre B si solo si cumple *continuing down* sobre B.[1]

3 Un Algoritmo para Safe Contraction

3.1 ¿Cuál es el problema de la fuerza bruta?

Analizando la definición de la función Safe Contraction se observa que es necesario encontrar todas las demostraciones minimales de la fórmula por la que se contrae (en adelante f) dentro de la teoría (en adelante K). Podemos pensar a cada demostración como un elemento de $P(K)$. Un algoritmo simple para contracción usando Safe Contraction podría examinar uno a uno los elementos de $P(K)$, verificando en cada caso si es una demostración minimal de f. Luego,

aplicando el orden Safe a cada una de estas demostraciones, identificaría los elementos que no sean safe y los eliminaría de la teoría. Por último, clausurando los elementos safe, se obtendría el resultado final $K - f$.

El algoritmo para contracción podría ser como sigue:

1. Ingresar la teoría
2. Ingresar la fórmula f por la que se contrae.
3. Ingresar el orden safe para la teoría
4. Para cada elemento de $P(K)$ hacer:
    5. Si es demostración minimal de f.
        6. Marcar los elementos que no sean safe.
7. Obtener K/f: los elementos safe de K (los elementos que no fueron marcados en ninguna demostración)
8. Resultado: $Cn(K/f)$.

Si la teoría K tiene m fórmulas, el conjunto $P(K)$ tendrá 2^m elementos.

En el siguiente cuadro se muestran ejemplos de la relación entre la cantidad de elementos de K y la cantidad de elementos de $P(K)$ trabajando con tres variables proposicionales. Además se indica la cantidad de demostraciones minimales que existen en cada caso.

K	f	$\#K$	$\#P(K)$	# dem. minimales
$Cn(\{a, -a\})$	$a \wedge \neg a$	256	2^{256}	$3.731.508 < 2^{22}$
$Cn(\{a, b, c\})$	$a \wedge b \wedge c$	128	2^{128}	$129.425 < 2^{17}$
$Cn(\{a, b, c\})$	$a \vee b \vee c$	128	2^{128}	$64 = 2^6$

Cuadro 2

Más adelante se mostrará que el algoritmo que presentamos en este trabajo evalúa para el tercer caso apenas 128 conjuntos para encontrar las 64 demostraciones minimales, es decir, 2^7 conjuntos que es un número notablemente menor a 2^{128} que se obtendrían por fuerza bruta, lo cual resultaría intratable.

3.2 Nuestro Algoritmo basado en heurísticas

El algoritmo obtiene todas las demostraciones minimales de f en K, aplicando diferentes heurísticas para acotar drásticamente el número de elementos de $P(K)$ examinados.

El algoritmo va generando elementos de $P(K)$ que son conjuntos de fórmulas candidatos a ser demostraciones minimales de f. Se construyen de manera incremental sobre la cantidad de elementos utilizando técnicas de backtracking para realizar una búsqueda exhaustiva.

Uno de los elementos principales en la reducción del espacio de cómputo es la existencia de una cota para la cantidad de fórmulas que pueden participar en una demostración minimal de f. Esta cota depende de la cantidad de variables proposicionales con que se esté trabajando y se puede mejorar aún más, teniendo en cuenta la fórmula f por la que se quiere contraer.

Más adelante se detallará cómo esta mejora se logra analizando la tabla de verdad de la fórmula f.

El algoritmo genera cada demostración partiendo del conjunto vacío y agregando de a una fórmula hasta obtener una demostracion minimal. Pero sólo se agrega una fórmula cuando esta resulta "útil", es decir, cuando realmente aporta nueva información a la demostración que se está elaborando. Por ejemplo si D es un conjunto candidato a ser demostración y g es una nueva fórmula que aún no pertenece a D, si $Cn(D) = Cn(D \cup \{g\})$ entonces g no es relevante y por lo tanto no se agregaría al conjunto D.

3.2.1 Estructuras de datos

Una de las principales estructuras que se utilizan es $TablasV$, que es un vector que tiene una posición por cada fórmula posible. $TablasV[i]$ representa la tabla de verdad de la fórmula i. Dado que estamos trabajando con clases de equivalencia y con cantidad finita de variables, este vector contiene una cantidad finita de elementos. Cada bit j de $TablasV[i]$ representa el valor de verdad de la fórmula i para la valuación j (donde usamos 1 para verdadero y 0 para falso).

Usamos además otras dos estructuras con formato similar que son *Base* y *Teoría*. *Base* contiene las tablas de verdad de las fórmulas ingresadas por el usuario, y *Teoría* contiene las tablas de verdad de la teoría asociada a *Base*.

Otra estructura de datos fundamental es *Demostración*. Es un vector en dónde se van almacenando elementos de *Teoría* para poder evaluar si el conjunto de fórmulas que representa constituye una demostración minimal de f.

El orden $\prec$ elegido se almacena en una matriz *Orden* donde $Orden[i, j]$ es verdadero sii $Teoria[i] \prec Teoria[j]$.Para verificar que cumple la propiedad de continuing up se emplea además una matriz *Implica* donde $Implica[i, j]$ es verdadero sii $Teoria[i] \rightarrow Teoria[j]$.

TextosFormulas es la estructura que se usa para relacionar las tablas de verdad y el texto de la fórmulas.

El algoritmo va formando conjuntos de fórmulas como candidatos a ser una demostración de f. En determinadas ocasiones se detecta que hay alternativas que no necesitan ser exploradas. En estos casos se cancela la búsqueda con el conjunto parcial obtenido y decimos que se corta un camino (que daría origen a múltiples conjuntos de fórmulas a analizar). Para cubrir todas las altenativas posibles se emplea la técnica de backtracking y se utiliza una pila P que lleva cuenta de los caminos recorridos. Cuando un camino es cancelado se utiliza P para identificar el próximo conjunto a explorar.

En P se guarda tanto la posición de la fórmula dentro de *Teoria* como la cantidad de fórmulas que se habían acumulado en *Demostración* hasta ese momento. Los elementos de P son pares con dos componentes: *Indice* y *Longitud*. *Indice* es la posición, dentro de *Teoría*, de la última fórmula visitada y *Longitud* representa la cantidad de fórmulas que se habían incluído en *Demostración*.

3.2.2 Detectando demostraciones

Verificar que $D \vdash f$ es equivalente a ver que $\vdash (\wedge D \rightarrow f)$. Es decir que para cada valuación de las variables proposicionales $\wedge D \rightarrow f$ es verdadero. Podemos reemplazar la expresión $\wedge D \rightarrow f$ por $\neg(\wedge D) \vee f$. En aquellas valuaciones donde f es verdadero, la expresión $\neg(\wedge D) \vee f$ es también verdadera. Para las valuaciones donde f es falso, la expresión $\neg(\wedge D)$ deberá ser verdadera, y esto ocurrirá únicamente si $\wedge D$ es falso para esa misma valuación.

En nuestra representación tomamos verdadero como un 1 y falso como un 0.

De este modo, con las estructuras mencionadas, *Demostración* es una demostración de f si para cada valuación donde f es 0 (falso), alguno de los elementos de *Demostración* también es 0, con lo cual $\wedge$*Demostración* también será 0 para esa valuación. Podemos decir que *Demostración* "cubre" los ceros de f, por lo tanto buscar una demostración para f es buscar un conjunto de fórmulas que cubran todos los ceros de f.

El siguiente cuadro ejemplifica la estructura de $TablasV$ y los valores asociados a cada fórmula posible en el caso de 2 variables proposicionales.

	Valuaciones																	
	a	b	⊥	-a ^ -b	a ^ -b	-b	-a ^ b	-a	-a <-> b	-av-b	a ^ b	a <-> b	a	av-b	b	-avb	avb	T
1	F	F	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	V	F	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
3	F	V	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
4	V	V	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Cuadro 3

En el cuadro mostrado se puede observar que $D = \{a, b\}$ es una demostración de $a \wedge b$. La fórmula $a \wedge b$ tiene valor 0 en las valuaciones 1, 2 y 3, se puede ver que el conjunto D "cubre" estos ceros, ya que para la valuación 3 a tiene un 0, para la valuación 2 b tiene un 0 y para la valuación 1 tanto a como b tienen 0.

3.2.3 Heurísticas aplicadas

En esta sección se comentan las principales heurísticas utilizadas y las razones en que se basan.

1) Buscar demostraciones sólo si es necesario.

Teniendo en cuenta que se pueden realizar operaciones de expansión, contracción y revisión, no en todos los casos corresponde hacer una búsqueda de las demostraciones.

Para el caso de expansión nunca es necesario hacer el cálculo de las demostraciones porque simplemente se agregan fórmulas a la teoría original sin realizar ninguna contracción. En el caso de la contracción, sólo se buscaran demostraciones cuando la fórmula f se deduzca de la teoría. Utilizando la identidad de Levi la revisión se realiza en dos pasos: primero contrayendo por $\neg f$ y luego

expandiendo por f. Por lo tanto sólo se buscarán demostraciones cuando $\neg f$ se deduzca de la teoría.

Cuando no se necesitan calcular las demostraciones, tampoco hace falta pedir el orden, ya que este se utiliza para obtener los mínimos en las demostraciones, con lo cual se evitan pasos innecesarios.

2) Cota para la cantidad de elementos en una demostración minimal

Si tenemos en cuenta que buscar una demostración para f es equivalente a encontrar un conjunto D de fórmulas que cubran los ceros de f para cada valuación, en el caso límite en que f tenga 0 en todas sus valuaciones, el conjunto D deberá cubrir tantos ceros como valuaciones haya. Por otra parte, para que la demostración sea minimal, cada fórmula g de D debe aportar al menos un cero que no se pueda obtener del resto de las fórmulas en D, pues de lo contrario quitando g igualmente se cubrirían todos los ceros de f.

De esto se deduce que en el caso extremo en una demostración se necesitarán tantas fórmulas como valuaciones existan, cada una aportando exactamente un cero para la demostración de f. Como la cantidad de valuaciones depende de la cantidad de variables, una demostración minimal requerirá a lo sumo 2^n fórmulas, siendo n la cantidad de variables proposicionales.

En el siguiente cuadro se muestra la relación entre la cantidad de variables y la cantidad de valuaciones y fórmulas posibles.

#Var	#Val. $(2^{\#Var})$	#Flas $(2^{\#Valuaciones})$	#max. de flas dem.
2	4	16	4
3	8	256	8
4	16	65536	16
5	32	4.294.967.296	32
6	64	1.84467E+19	64

Cuadro 4

3) Refinamiento para la cota según f

Si bien la cota anterior reduce notablemente la cantidad de subconjuntos a considerar como demostraciones, no hace uso de la información que aporta la fórmula f. Haciendo un simple análisis de la tabla de verdad de f, se puede reducir aún mucho más la cantidad de casos a analizar.

Utilizando la misma idea mencionada anteriormente de que el conjunto D debe cubrir todos los ceros de f y que cada fórmula debe aportar al menos un cero distinto, la cantidad máxima de fórmulas en una demostración, es la cantidad de ceros de la fórmula f. Esto significa que la cota anterior sólo aplicaría cuando la fórmula f sea la contradicción, que contiene ceros en todas sus valuaciones (peor caso).

4) Criterios para agregar una fórmula a una demostración.

Como mencionáramos anteriormente, sólo se agrega una fórmula a una demostración cuando es relevante, es decir, cuando aporta un nuevo 0 de f. De esta manera, haciendo un simple chequeo de la tabla de verdad de las fórmulas,

podemos detectar aquellas fórmulas irrelevantes que no aportan ningún cero nuevo de f a la demostración que se esta generando.

5)Evitando caminos que sólo conducen a demostraciones no minimales.

Teniendo en cuenta que las demostraciones de $m+1$ elementos se obtienen después de haber buscado todas las demostraciones de m elementos y sabiendo que cada fórmula debe aportar al menos un cero distinto de f, se pueden detectar y evitar caminos que llevan a demostraciones que resultarán no minimales.

Cuando se están generando demostraciones de m elementos y se llega a una demostración antes de incorporar m fórmulas el camino es descartado pues al agregar alguna fórmula más no se podría obtener una demostración minimal de f. Este caso puede ser detectado antes de obtener una demostración: se puede comparar la cantidad de ceros que restan cubrir con la cantidad de fórmulas que faltan agregar para llegar a los m elementos. Si resulta ser que la cantidad de ceros es menor que la cantidad de fórmulas que faltan agregar, no tiene sentido continuar generando ese conjunto porque al llegar al elemento m podremos asegurar que la demostración no será minimal.

6) Ahorrando cálculo al controlar la minimalidad.

Para verificar si una demostración encontrada es minimal, lo que se hace es ir sacando de a uno por vez los elementos y verificando si con el conjunto resultante aún se implica f. Como la última fórmula se agregó a un conjunto que no implicaba f, no hace falta sacar esa fórmula y volver a comparar, con lo cual se evita el análisis de un caso por cada demostración encontrada.

7) Evitando caminos que no conducen a demostraciones de la cantidad de elementos buscada.

Cuando se están buscando demostraciones de m elementos y al comenzar a armar el conjunto la cantidad de fórmulas que restan analizar es menor a m, no tiene sentido continuar porque no se llegará a armar una demostración con m elementos.

8) Reducción de la cantidad de combinaciones a analizar.

Como dijimos anteriormente, las demostraciones de m elementos se generan antes que las de $m+1$. Así, las demostraciones de un único elemento se generan al principio del algoritmo. Cómo se buscan las demostraciones minimales, dichos elementos no pueden participar en ninguna demostración de más de un elemento porque no serían minimales, por lo tanto se marcan para que no sean considerados en las búsquedas de demostraciones de más de un elemento, reduciendo así la cantidad de combinaciones de fórmulas a analizar.

9) Si el orden es vacío no es necesario hacer la clausura del conjunto K/f.

Si el orden es vacío tenemos que $K - x = K/x$ y no es necesario calcular $Cn(K/x)$.

Sea D una demostracion minimal de x, ninguno de los elementos de D es safe dado que todos resultan ser mínimos porque el orden es vacío. Además por ser D minimal, $D - \{y\}$ no implica x para ningún $y \in D$.

Queremos demostrar que $y \in D$ no se implica por ningún conjunto de elementos safe $\{g_1, ..., g_n\}$.

Supongamos que $\{g_1, ..., g_n\}$ implica y, entonces si en la demostración D reemplazamos y por $\{g_1, ..., g_n\}$obteniendo D', $\exists\ D'' \subseteq D'$ tal que D'' es una demostracion minimal de x y $D - \{y\} \subset D''$. Pero entonces $\exists\ g_i \in D''$ que participa en una demostración minimal de x y como el orden es vacío $g_i \notin K/x$. Absurdo pues $g_i \in K/x$ por hipótesis.

Este resultado ya había sido obtenido en observación 3.4 en [1].

3.2.4 Pseudocódigo del algoritmo propuesto

1- Ingresar la base

2- Ingresar la fórmula f por la cual se opera y seleccionar la operación a realizar.

3- Obtener la teoría K asociada a la base ingresada

4- Determinar si es necesario calcular las demostraciones.

5- Si es necesario calcular demostraciones

a. Pedir el orden safe para K.

b. Obtener demostraciones minimales de f incluídas en K.

c. Seleccionar los elementos "no safe" de cada demostración.

6- Calcular $K - f$ (obtener K/f y clausurar).

3.2.5 Funciones principales

Funcion ObtenerTeoria

Entrada:

$CantVar$: es un entero que indica la cantidad de variables proposicionales de la base

$Formulas$: es una lista de strings con las fórmulas de la base y la fórmula a operar en la última posición.

Salida :

$Teoria$: vector de enteros,

$AlfaEstaEnBase$, $NoAlfaEstaEnBase$: booleano

Esta función obtiene la teoría asociada a la base ingresada ($Formulas$). Para ello evalúa las fórmulas ingresadas (incluyendo la fórmula por la cual se opera) para cada valuación y obtiene el vector $Base$ con la tabla de verdad de cada una de las fórmulas. Luego se obtiene la teoría, para lo cual se hace un AND entre todas las fórmulas de la base, obteniendo una tabla de verdad (llamada $Patron$) que sintetiza la base. Para llegar a la teoría se seleccionan de $TablasV$ todas las fórmulas que son implicadas por $Patrón$. En esta función también se determina si la fórmula por la que se opera o su negación se deducen de la base. EvaluarFormulas evalúa cada fórmula y verifica su sintaxis a través de un parser de precedencia. [9]. Por último se construye la matriz $Implica[]$, donde $Implica[i, j] = True$ cuando la fórmula $Teoria[i]$ implica a la fórmula $Teoria[j]$.

Para i=0 hasta #Formulas-1 hacer

Para j=0 hasta $2^{CantVar} - 1$ hacer

Si EvaluarFormulas(i, Formulas[j]) // si la fla i es true para la val j

Base[i]=Base[i] or 2^j //activo el bit de true

FinPara
FinPara
Para i=0 hasta #Formulas-2 hacer //se excluye a la fórmula f
Patron = Patron and Base[i] // AND de todas las flas de la base
FinPara
{Se determina si la fórmula f ó su negación se deducen de la base, para determinar luego si corresponde hacer el cálculo de las demostraciones}
AlfaEstaEnBase = ((Patron and f) = Patron) // ver si $base \vdash f$
NoAlfaEstaEnBase = ((Patron and not f) = Patron) // ver si $base \vdash \neg f$
{Se analizan todas las fórmulas posibles
para ver cuáles se deducen de la base}
Para i=0 hasta $2^{2^{CantVar}} - 1$ hacer
Si ((Patron and TablasV[i]) = Patron) // si la base $\vdash$ TablasV[i]
Incorporar TablasV[i] a Teoria
FinPara
{Se construye la matriz Implica}
Para i=1 hasta #Teoria hacer
Para j=1 hasta #Teoria hacer
Implica[i, j] = ((Teoria[i] or Teoria[j]) =Teoria[j])
FinPara
FinPara

Función ObtenerDemostraciones
Entrada:
$CantVar$:es un entero que indica la cantidad de variables proposicionales de la teoría
f: es la fórmula para la cual se quieren encontrar demostraciones.
Salida:
Demostraciones: lista de demostraciones minimales
Minimos: vector con las fórmulas que resultaron ser mínimas en alguna demostracion minimal de f según el orden safe.
Demostracion es una variable local que mantiene la demostración actual de f.

El algoritmo arma primero todas las demostraciones de un elemento, después las de dos y así hasta llegar a la cota (cantidad de ceros de f).

Se arma un conjunto de i elementos (cuando estamos en la pasada i) candidato a ser demostración y luego se verifica si es minimal. En caso de no serlo se descarta.

Se intentan armar demostraciones combinando todos los elementos de la teoría. Para optimizar la búsqueda se aplican las diferentes heurísticas enunciadas anteriormente. Se utiliza la técnica de backtracking y estructuras de pilas para recorrer todos los caminos. Una vez que se encuentra una demostración minimal, se le calculan los mínimos según el orden elegido y se conservan para ser quitados de la teoría más tarde.

La idea del algoritmo es la siguiente: primero se obtienen todas las demostraciones de un elemento. Los elementos que resulten ser demostraciones para f.

no se utilizarán en la búsqueda de demostraciones de más de un elemento porque esas demostraciones no podrían ser minimales.

En la pasada de 2 elementos se combina cada uno de los elementos de la teoría (que no fueron demostraciones en la pasada de 1 elemento) con cada uno de los elementos que se encuentran a su derecha en el vector y se verifica si es una demostración. En ese caso no hace falta verificar minimalidad porque se puede asegurar que cada uno de los elementos por separado no era demostración. Este es el motivo por el cual se trató por separado la pasada de 2 elementos.

Para la pasada de más de 2 elementos (caso general) se hace lo siguiente:

Para lograr demostraciones minimales de i elementos, se van tomando sucesivamente $i-1$ fórmulas como pivotes y se van combinando con los elementos que se encuentran siempre a la derecha en el vector. De esta forma se puede garantizar que no se genere ningún conjunto más de una vez en la misma pasada. Para que una fórmula sea incorporada al conjunto candidato a ser demostración debe cumplir con lo siguiente:

- debe cubrir un nuevo cero de f. *(Heurística 4)*

- la cantidad de ceros cubiertos por la fórmula a incorporar, junto con el conjunto generado hasta el momento, no debe ser superior a la cantidad de fórmulas que faltan agregar para llegar a formar una demostración de i elementos. *(Heurística 5)*

- si al agregar la fórmula llegara a una demostración de f, pero aún no se obtuvieron i elementos en el conjunto, se descarta la fórmula y se pasa a la siguiente, porque esa demostración ya se obtuvo en alguna pasada anterior de menos elementos. *(Heurística 5)*

- un conjunto de i elementos que no es demostración se descarta.

- si al agregar la fórmula se llegara a una demostración de i elementos, se verifica si dicha demostración es minimal. Si no lo fuera se descarta la fórmula y se pasa a la siguiente, porque sólo se buscan demostraciones minimales.

Cuando un pivote no se puede combinar con más fórmulas hacia la derecha, se hace backtracking para buscar el siguiente pivote. Si no se pudiera encontrar otro pivote hacia la derecha se hace backtracking al pivote anterior, y así sucesivamente hasta que no se puedan generar nuevos pivotes, y en ese caso termina la búsqueda.

Unicamente pueden ser candidatos a ser primer pivote las fórmulas que tengan al menos $i-1$ fórmulas a su derecha en el vector, porque de lo contrario no podrían llegar a generar conjuntos de demostraciones de i elementos. *(Heurística 7)*

{Se obtiene la cantidad de ceros de f para determinar la cota principal del algoritmo. Para cada valuación se analiza si f tiene un uno.}

Para i=1 hasta #Valuaciones hacer

  Si (((1shl i) and f) <> 0) //si el bit i de f es uno

    nivelf = nivelf +1 //incremento la cant. de unos de f

FinPara

{En base a la cantidad de unos de f se obtienen la cantidad de ceros que son necesarios "cubrir"}.

#MaximaElementos=#Valuaciones-nivelf.

```
Para i = 1 hasta #MaximaElementos hacer
    Segun i hacer:
    Si i=1: //demostraciones de 1 elemento
        {Para cada elemento de la teoría se verifica si implica f, en
        cuyo caso es demostración, de lo contrario se guarda en
        TeoriaAux para la búsqueda de demostraciones de más
        de un elemento. (Heurística 8)}
        Para j=1 hasta #Teoria hacer
            Demostracion = {TeoriaAux[j]}
            Si ( f or teoria[j] ) = f
                detectarMinimos(demostracion)
                Demostracion = Ø
            Sino        //si no es demostración
                {Se aplica la heurística 8.}
                GuardarElemento(TeoriaAux, Teoria[j])
            FinSi
        FinPara
    Si i=2:        //Demostraciones de dos elementos
    Demostracion = Ø
    {Se analizan todos los pares de TeoriaAux, si
    algún par es demostración también es minimal}
    Para j=1 hasta #TeoriaAux hacer
        Para k= j+1 hasta #TeoriaAux hacer
            Si ( f or (TeoriaAux[j] and TeoriaAux[k])) = f
                Demostracion = {TeoriaAux[k], TeoriaAux[j]}
                detectarMinimos(Demostracion)
                Demostracion = Demostracion - {TeoriaAux[k]}
            FinSi
        FinPara
    FinPara
    Si i > 2:    //búsqueda de demostraciones de más de dos elementos
        Para j=0 hasta (#TeoriaAux -i+1) hacer      //aplicamos heurística 7
            Demostracion = Ø
            {Solo se puede agregar un elemento a la
            demostración si cubre un cero nuevo de f (Heurística 4)}
            Si (not f and ¬TeoriaAux[j] ) <> 0
                {Se determina si tiene sentido seguir con la demostración
                actual en función de los ceros obtenidos y
                faltantes (Heurística 5)}
                Si ¬ ( tieneSentidoSeguir(TeoriaAux[j], Demostracion))
                    saltar a la próxima iteración
                FinSi
                Demostracion={TeoriaAux[j]}
                k = j + 1      //se recorre hacia la derecha
                Mientras k < #TeoriaAux hacer
                    {Se aplica la heurística 4}
```

```
                    Si aportaUnCeroNuevo(Demostracion, TeoriaAux[k], f )
                        {si es una demostración pero tiene menos
                         elementos que los buscados, se descarta la fórmula
                         y se pasa a la siguiente. Se aplica la heurística 5}
                        Si ( f or ( ΛDemostracion and TeoriaAux[k]))=f
                            {si es la cantidad de elementos buscada}
                            Si #demostracion+1= i
                                Demostracion = Demostracion ∪{TeoriaAux[k]}
                                Si esMinimal(demostracion, f)
                                    detectarMinimos(demostracion)
                                FinSi
                                Demostracion = Demostracion - {TeoriaAux[k]}
                            FinSi
                        Sino // el conjunto todavía no implica f
                            {Si hay mas candidatos se apila siempre
                             y cuando se pueda llegar a i elementos}
                            Si (#demostracion+1 < i) and (k < High(TeoriaAux))
                                Demostracion = Demostracion ∪{TeoriaAux[k]}
                                candidato = (ΛDemostracion and TeoriaAux[k])
                                Si¬(tieneSentidoSeguir(candidato,Demostracion))
                                 Demostracion = Demostracion - TeoriaAux[k]
                                Sino      //es un candidato válido
                                {Se apila el candidato y la cant. de elem.
                                 de Demostracion para el backtracking}
                                    Aplilar(P, (k, #Demostracion))
                                FinSi
                            FinSi
                        FinSi
                    FinSi
                    {Si se llega al final desapilar hasta encontrar otro pivote.
                    Si la pila está vacía, terminar}
                    Si (k = #TeoriaAux) and ¬ pilaVacia(P)
                        k = tope(P).indice
                        size = tope(P).longitud
                        {se toman los primeros size elementos.}
                        setLength (Demostracion, size)
                    FinSi
                    k = k + 1
                FinMientras
            FinSi
        FinPara
    FinSi
  FinSegun
 FinPara
Fin
```

Función tieneSentidoSeguir
Entrada:
candidato es la fórmula candidata a ser agregada a D
D la demostración actual.
Salida:
resultado es un booleano que indica si tiene sentido agregar la fórmula a D.
{obtener acumulado sin contar unos de f}
acumulado = candidato or f
cantCeros =0
cantFormulasFaltantes = 0
{contar los ceros de f obtenidos hasta el momento}
Para m = 1 hasta #Valuaciones hacer
Si (((1 shl m) and acumulado) = 0)
cantCeros = cantCeros + 1
FinPara
{Se obtiene la cantidad de ceros de f que
faltarían cubrir si agregara candidato a D}.
cantCerosFaltantes = cantMaximaElementos - cantCeros
{Se determina la cantidad de fórmulas que
faltan agregar para llegar a n elementos}
cantFormulasFaltantes = i - #D
{Si la cantidad de ceros que faltan es menor que la cantidad de fórmulas que
faltan para llegar a i elementos, no tiene sentido intentar con esta fórmula
y combinarla con los elementos que están a su derecha (heurística 5)}
Si cantCerosFaltantes < cantFormulasFaltantes
resultado := false
sino
resultado := true
FinSi

Función aportaUnCeroNuevo
Entrada:
D: es un conjunto candidato a ser demostración
candidato: es la fórmula candidata a ser agregada
f: es la fórmula para la cual se calculan las demostraciones
Salida:
resultado es un booleano que indica si candidato aporta un nuevo cero de f
Como sólo se verifican ceros de f, se incluye f a ambos lados para no comparar las valuaciones donde f es true.
Si ($(\Lambda D \wedge \text{candidato}) \vee f) \neq \Lambda D \vee f$
resultado = true
Sino
resultado = false
FinSi

Función SafeContraction

Entrada:
Minimos: vector con los elementos que resultaron ser mínimos en alguna demostración
f: es la fórmula por la que se contrae
Salida:
$K - f$

Esta función obtiene los elementos que no resultaron mínimos en ninguna demostración minimal según el orden Safe y los almacena en el vector Resultado. Si la operación elegida fue revisión o expansión, agrega también la fórmula por la que se está operando. Luego calcula la clausura de los elementos que se encuentran en Resultado, para ver si alguna fórmula que haya resultado mínimo en alguna demostración se deduce de los elementos safe. En el apéndice se muestra un ejemplo.

```
{Deja los NO menores en Resultado}
obtenerElementosSafe() //Resultado = Teoría - Mínimos
Si (gOperacion = 'R')
    agregarFormula(¬f)      //Agrega ¬f a Resultado
Si (gOperacion = 'E')
    agregarFormula(f)       //Agrega f a Resultado
{calcula el AND de todas las fórmulas que resultaron safe}
Para i= 1 hasta #Resultado hacer
    sintesisResultado = sintesisResultado and Resultado[i]
FinPara
resultado = Ø
{Se analizan todas las fórmulas en TablasV para
ver cuáles se deducen de los elementos safe}
Para i= 1 hasta #TablasV hacer
    Si (sintesisResultado and TablasV[i]) = sintesisResultado
        Incorporar TablasV[i] a Resultado
    FinSi
FinPara
```

Función ChequearOrden
Entrada:
Orden: Matriz con el orden definido por el usuario
Salida:
Orden: Matriz de orden actualizada con la transitividad
Result: es un entero que indica con 0 si hubo algún error al validar las propiedades o un 1 si el orden safe satisfizo las propiedades
En esta función se verifica que el orden elegido cumpla con las propiedades de aciclicidad y virtually connected y completa la matriz de Orden, reflejando las relaciones transitivas.

```
{Se aplica Warshall para la clausura transitiva}
Para i = 1 hasta #Formulas hacer
    Para j = 1 hasta #Formulas hacer
```

```
            Para k = 1 hasta #Formulas hacer
                {Si se verifica que la fórmula j es menor que la
                fórmula i, y la fórmula i es menor que la fórmula k,
                y además j=k, entonces significa que hay un ciclo
                en el orden, no se puede continuar. Sino, se guarda
                 la relación transitiva j<k en la matriz Orden}
                Si Orden[j,i] and Orden[i,k]
                    Si j=k then
                        Result = 0 // Ciclo en el orden
                        exit
                    FinSi
                Sino
                    Orden[j,k] = true
    {Se chequea virtual conectivity para cada orden: si se verifica que la
    fórmula i es menor que la fórmula j, pero existe una fórmula k que no
    es menor que la fórmula j ni mayor que la fórmula i, entonces el orden
    no cumple la propiedad de virtually connected.}
    Para i = 1 to #Formulas hacer
        Para j = 1 to #Formulas hacer
            Si Orden[i,j]
                Para k = 0 to #Formulas hacer
                    Si ((not Orden[i,k])and(not Orden[k,j]))
                        Result = 0 // El orden no cumple VC
                        exit
                    FinSi
                FinPara
            FinSi
        FinPara
    FinPara
    Result = 1
```

Función ChequearContinuing
Entrada:
Orden: Matriz con el orden.
Implica: Matriz con las relaciones de implicación entre cualquier par de fórmulas.
Salida:
Result: es un entero que indica con 0 si hubo algún error al validar la propiedad o un 1 si el orden safe satisfizo las propiedades

En esta función se verifica que el orden elegido cumpla con la propiedad de continuing up. Dado que en presencia de virtually connected si se cumple continuing up se puede garantizar que se cumple continuing down, sólo se verifica continuing up.

```
    {Se verifica si cumple Continuing Up}
    cumple = true
    Para i= 1 hasta #Orden hacer
```

```
    Para j= 1 hasta #Orden hacer
        {Si la fórmula i es menor que la fórmula j se buscan
        todas las fórmulas k que se deduzcan de la fórmula j y
        se verifica si la fórmula i es menor que la fórmula k.}
        Si Orden[i,j] // Si la fórmula i es menor que la fórmula j
            {Se buscan todas la fórmulas que sean implicadas por j}
            Para k= 1 hasta #Implica hacer
                Si Implica[j,k] and not Orden[i,k] // i < k ?
                    cumple = false
                    break
                FinSi
            FinPara
        FinSi
        Si not cumple
            Break
        FinSi
    FinPara
    Si not cumple
        Break
    FinSi
FinPara
Si cumple
    Result = 1
Sino
    Result = 0
FinSi
```

3.3 Observaciones sobre las funciones principales

El problema a resolver es inherentemente NP. Algunos subproblemas dentro de nuestro algoritmo son problemas NP-completos. Por ejemplo para obtener la teoría con la que trabajamos debemos identificar las valuaciones que satifacen a la todas las fórmulas de la base ingresada, lo cual es un problema similar a SAT [6]. La estructura $TablasV$ almacena la tabla de verdad de todas las fórmulas posibles para un número de n variables proposicionales, lo cual da origen a 2^{2^n} fórmulas lógicamente distintas. El solo hecho de generar esta tabla es un problema exponencial respecto del número de variables proposicionales empleadas. La clausura de K/f tiene complejidad lineal respecto de la cantidad de elementos de $TablasV$, pero dado que la longitud de este vector es 2^{2^n}, el problema se transforma en exponencial.

La búsqueda de demostraciones requiere un comentario aparte. Es un problema NP respecto de la cantidad de fórmulas en la teoría. Cuando la base ingresada es insatisfactible, la teoría asociada es $\mathcal{L}$. En este caso es fácil de ver que cualquier fórmula g participará en una demostración de cualquier fórmula f ($f \neq \mathsf{T}$) por la que se desee contraer o revisar. Si $g \vdash f$ entonces $\{g\}$ es una demostración minimal, en otro caso se puede formar una demostración minimal

incluyendo la negación de g: $\{g, \neg g\} \vdash f$. La mayor cantidad de demostraciones se obtiene cuando la fórmula f es también insatisfactible. Al margen del cómputo requerido para detectar las demostraciones, el hecho de listarlas es de por sí un problema NP.

Búsqueda de demostraciones

El algoritmo genera conjuntos candidatos a ser demostración comenzando por conjuntos con una fórmula y luego de dos, tres, etc. hasta llegar a una cota que depende de la cantidad de valuaciones en que la fórmula f es falso, es decir la cantidad de ceros de f. En el peor caso para n variables proposicionales la fórmula es siempre falsa (contradicción) y la cantidad de ceros es 2^n, por lo que se generarán conjuntos de hasta 2^n fórmulas. Además los conjuntos se construyen con fórmulas de la teoría que tiene 2^{2^n} elementos en el peor caso. Con todo esto podemos decir que la complejidad de esta función es exponencial respecto de la cantidad de variables proposicionales que se empleen.

Búsqueda de mínimos según el orden safe

Una vez obtenida una demostración, se búscan los elementos que no son safe según el orden introducido. Esta función toma cada fórmula g_i en la demostración y busca alguna otra fórmula g_j que sea menor que g_i con respecto al orden safe.Si la fórmula g_j no es encontrada, entonces g_i es un mínimo y por lo tanto no es un elemnto $safe$. En el peor de los casos, con m fórmulas en una demostración, todas las fórmulas son mínimos y nunca se encuentra la fórmula g_j con lo que se recorre toda la demostración por cada fórmula, dando un orden de m^2.

Clausura de K/f

En la función $SafeContraction$ se identifican los elementos safe como aquellas fórmulas que no son mínimos en ninguna demostracíon haciendo una resta de conjuntos entre la teoría y los mínimos detectados. Este cálculo tiene una complejidad de m^2 (con m fórmulas en la teoría).

Luego se clausuran los elementos $safe$ examinando todas las fórmulas de $TablasV$ para verificar si son implicadas por los elementos $safe$. Para lograr esto primero se aplica a todos los elementos safe el operador lógico $\wedge$ obteniendo una fórmula s que sintetiza lo que los elementos $safe$ en conjunto implican. Finalmente se revisan todas las fórmulas posibles para ver si son implicadas por s. En el peor caso todos los elementos son safe y la cantidad de elementos de la teoría es $\mathcal{L}$, con lo que la complejidad es de $m+m$.

3.3.1 Comparación de diferentes formas de organizar las tablas de verdad

Se explicaron anteriormente diferentes heurísticas para que el problema sea tratable para un n pequeño. Sin embargo existe un punto adicional que tiene un impacto notable en el cálculo: la forma en que almacenan las fórmulas en la estructura $Teoria$.

El algoritmo guarda en $Teoria$ aquellas fórmulas de $TablasV$ que son implicadas por la base ingresada. Al extraer estas fórmulas no se altera el orden que tenían en $TablasV$. Luego se recorren de izquierda a derecha intentando

encontrar demostraciones de f como ya se explicó. Cada fórmula que se agrega al conjunto candidato a ser demostración debe ser relevante para implicar a f. Cuando una demostración es hallada se verifica si es una demostración minimal.

Un ejemplo típico de demostración no minimal es cuando la última fórmula g_i agregada a la demostración implica a alguna fórmula g_j ya incorporada. En este caso g_j no es necesaria para implicar a f. Podemos decir que g_i aporta más información para implicar a f que g_j. El algoritmo identifica como relevante a una fórmula si cubre al menos un cero de f que no era cubierto por las fórmulas anteriores en conjunto. En el ejemplo mencionado g_i cubre los mismos ceros que cubría g_j e incluso alguno más. Este es sólo un ejemplo para entender la idea pero no es necesario en realidad que g_i implique a g_j porque sólo se revisan las valuaciones donde f es falso. En las valuaciones donde f es verdadero podría darse que g_i fuera verdadero y g_j no, con lo cual g_i no implicaría a g_j.

Es importante mencionar además que el algoritmo cancela la búsqueda cuando encuentra una demostración antes de tiempo. Si se están buscando demostraciones de cinco elementos y al incorporar el segundo ya se ha obtenido una demostración, ese camino es cancelado. La heurística va un poco más alla en realidad. Si al incorporar un elemento detecta que no se puede llegar a una demostracion de la cantidad de fórmulas planeadas para esa iteración, la búsqueda no continúa con ese conjunto.

En resumen si las fórmulas que más ayudan a implicar a f se encuentran más a la izquierda de la estructura, más rápido se encontrará una demostración y más caminos que no llevan a demostraciones minimales serán detectados antes de haber agregado demasiados elementos a un conjunto candidato.

El siguiente cuadro muestra la cantidad de demostraciones minimales encontradas para diferentes combinaciones de bases (filas) y fórmulas (columnas).

Base	⊥	a^b^c	a^b	a^(bvc)	a	av(b^c)	avb	avbvc	T	#K
⊥	3.731.508	2.083.554	1.052.500	460.936	162.064	38.944	4.160	128	-	256
{a,b,c}	-	129.425	74.662	38.628	16.968	5.648	1.056	64	-	128
{a,b}	-	-	6.424	3.874	2.068	904	272	32	-	64
{a, (bvc)}	-	-	-	462	294	164	72	16	-	32
{a}	-	-	-	-	49	34	20	8	-	16
{av(b^c)}	-	-	-	-	-	8	6	4	-	8
{avb}	-	-	-	-	-	-	2	2	-	4
{avbvc}	-	-	-	-	-	-	-	1	-	2
T	-	-	-	-	-	-	-	-	-	1

Cuadro 5

Se aprecia en el cuadro cómo aumenta exponencialmente la cantidad de demostraciones a medida que aumenta la cantidad de fórmulas en la teoría.

Este cuadro es independiente del ordenamiento que se elija para Teoria. Las diferentes estrategias llegarán más rápido o más lento a la solución, pero en cualquier caso las demostraciones minimales serán encontradas. Al cambiar la forma de organizar las tablas de verdad se aprecian tiempos de ejecución

notablemente diferentes. En general al colocar las fórmulas lógicamente más fuertes al comienzo, se obtienen tiempos de ejecución menores.

Nuestra implementación provee cuatro formas diferentes de organizar las tablas de verdad en la estructura $TablasV$:

1. *Mayor poder deductivo*: Las fórmulas con más cantidad de ceros en sus valuaciones al principio (a la izquierda).
2. *Menor poder deductivo*: Inverso al anterior. En primer lugar se encuentra la tautología y en último la contradicción.
3. *Por Identificador de fórmula*: La tabla de verdad de cada fórmula vista como un entero se utliza para clasificar las fórmulas
4. *Por Identificador de fórmula descendente*: Inverso al anterior.

El siguiente cuadro compara la cantidad de conjuntos generados y cancelados por las heurísticas con el ordenamiento *Mayor poder deductivo* y *Menor poder deductivo.*

Base	f	#K	#Dem	Conj. Generados		Conj. Cancelados por Heur.	
			Minimales	> poder deduct.	< poder deduct.	> poder deduct.	< poder deduct.
⊥	⊥	256	3.731.508	147.386.227	3.599.912.152	7.509.410	75.054.580
⊥	avbvc	256	128	256	256	-	-
a^b^c	a^b^c	128	129.425	2.354.733	8.865.870	177.676	550.103
a^b	a	64	2.068	29.655	41.364	316	764
a	avb	16	20	82	82	-	-
a^b	a^b	64	6.424	64.591	103.134	6.995	12.330
a^(bvc)	a^(bvc)	32	462	3.035	3.035	451	451
a	a	16	49	243	243	45	45
av(b^c)	av(b^c)	8	8	33	33	5	5
avb	avb	4	2	7	7	-	-
avbvc	avbvc	2	1	2	2	-	-
T	T	1	1	1	1	-	-

Cuadro 6

El ordenamiento de tablas de verdad tiene un enorme impacto en la cantidad de conjuntos generados. Por conjunto generado se entiende aquellos conjuntos de fórmulas que llegaron a incluir la cantidad de fórmulas requeridas para una iteración. Incluye a las demostraciones minimales, las que resultaron no ser minimales y a los conjuntos que no llegaron a ser demostración de f. Por conjunto cancelado se entienden aquellos conjuntos en los cuales se detiene la búsqueda antes de incorporar la cantidad de fórmulas requeridas para una iteración. Se puede ver en el cuadro que para el ordenamiento *Menor poder deductivo* se cancelan en general más búsquedas que para el de *Mayor poder deductivo* y que la cantidad de conjuntos generados tiene una relación similar. La pregunta que surge inicialmente es ¿cómo es posible que cortando más caminos se generen más conjuntos? La razón principal es que en el caso de *Mayor poder deductivo*

las demostraciones se encuentran con pocos elementos y se cortan caminos anticipadamente que podrían dar origen a muchas demostraciones no minimales. En el ordenamiento *Menor poder deductivo* las demostraciones que no cumplen con la cantidad pautada de elementos son detectadas sólo cuando ya se han incorporado muchas fórmulas al conjunto a ser demostración.

Para finalizar con la comparación de las diferentes formas de ordenar las tablas de verdad se presenta un cuadro ilustrando la cantidad total de conjuntos explorados con tres estrategias distintas. Por conjuntos explorados se entiende aquellos conjuntos que se generaron, independientemente de que haya sido o no demostración o que haya sido interrumpido por alguna heurística.

Base	*f*	#K	#Dem Minimales	Total Conjuntos explorados			P(K)
				> poder deduct.	< poder deduct.	Id Formula	
⊥	⊥	256	3.731.508	154.895.637	3.674.966.732	413.687.830	2^{256}
⊥	avbvc	256	128	256	256	256	2^{256}
a^b^c	a^b^c	128	129.425	2.532.409	9.415.973	4.188.038	2^{128}
a^b	a	64	2.068	29.971	42.128	39.361	2^{64}
a	avb	16	20	82	82	82	2^{16}
a^b	a^b	64	6.424	71.586	115.464	86.530	2^{64}
a^(bvc)	a^(bvc)	32	462	3.486	3.486	3.635	2^{32}
a	a	16	49	288	288	294	2^{16}
av(b^c)	av(b^c)	8	8	38	38	39	256
avb	avb	4	2	7	7	7	16
avbvc	avbvc	2	1	2	2	2	4
T	T	1	1	1	1	1	2

Cuadro 7

Se puede observar en el cuadro anterior que el ordenamiento Mayor poder deductivo es en todos los casos mejor o igual que Menor poder deductivo y que en algunos casos el Id de fórmulas llega a dar el peor resultado de los tres. Menor poder deductivo es en general un ordenamiento no deseable, pero no es siempre el peor caso.

4 Conclusiones y Trabajos futuros

El problema de buscar las demostraciones para una fórmula resulta generalmente intratable debido a la gran explosión combinatoria producida al intentar generar todos los posibles conjuntos, aún en un espacio de pocas variables proposicionales. En este trabajo presentamos un algoritmo que permite tratar un problema de complejidad exponencial en un tiempo razonable, basándose en diferentes heurísticas y tomando una cantidad reducida de variables proposicionales. La cota para la cantidad de variables se eligió de manera tal que fuese posible hacer una verificación y comparación de los resultados obtenidos.

El algoritmo tiene fines didácticos para poder analizar y explorar el comportamiento de la función Safe Contraction. Permite visualizar diversos elementos como ser la teoría asociada a la base ingresada por el usuario, las demostraciones obtenidas, como así también diversa información relacionada con el proceso de cálculo de las mismas, permitiendo evaluar y comparar distintas estrategias de búsqueda.

Dado que en general el orden Safe debe cumplir condiciones que no son fáciles de satisfacer a simple vista, en esta tesis se proponen algunos órdenes útiles y simples de generar que satisfacen todas las condiciones requeridas (ver apéndice).

Otra facilidad que provee es la posibilidad de seleccionar distintos criterios para ordenar $TablasV$, algunos de los cuales permiten disminuir el espacio de cómputo del algoritmo. En general ordenar $TablasV$ poniendo de izquierda a derecha las fórmulas con mayor poder deductivo (mayor cantidad de ceros) resulta más eficiente y poniendo las de menor poder deductivo primero (menor cantidad de ceros) en casi todos los casos resultó la peor opción. Si bien el algoritmo usa fuertemente la información de la fórmula por la que se contrae para acotar el espacio de búsqueda, queda pendiente analizar si dicha fórmula se podría aprovechar también para determinar un mejor ordenamiento de $TablasV$. En futuros trabajos también se podría analizar si el ordenamiento de $TablasV$ en base al orden Safe elegido permitiría un algoritmo más eficiente.

Durante la búsqueda de demostraciones no se aprovecha cuál es el orden Safe entre las fórmulas. Potencialmente se podrían agregar heurísticas que aprovechen esta información, o bien idear otro algoritmo que se base fuertemente en esta información para guiar la búsqueda.

Nuestro algoritmo trabaja con las clases de equivalencia de las fórmulas de la lógica proposicional. Un trabajo distinto hace falta para considerar fórmulas sintácticamente diferentes aunque lógicamente equivalentes. El orden Safe se hace, en general, más complicado en dichos casos.

Si bien la implementación que realizamos está acotada a un máximo de 3 variables proposicionales, sería factible extenderla a n variables, con poco impacto sobre el programa y las estructuras de datos. Principalmente habría que cambiar la representación de $TablasV$ para que se adapte a la cantidad de valuaciones posibles ($2^{2^{\#fórmulas}}$); y habría que completar la estructura que relaciona la tabla de verdad con los textos de cada fórmula.

Otro punto que se evidenció a lo largo de este trabajo, pero que requeriría una demostración formal, es el hecho de que, dada una base, la cantidad de demostraciones minimales para una fórmula queda determinada por la cantidad de ceros en la tabla de verdad de la base y la cantidad de ceros en la tabla de verdad de la fórmula.

5 Apéndices

5.1 Ejemplos de órdenes safe

Como se ha mencionado, elegir un orden que cumpla las propiedades requeridas por la función Safe Contraction es difícil y más aún trabajando con teorías. En particular la propiedad de *virtually connected* requiere que se expresen un número elevado de pares en la relación. Para simplificar esta tarea el algortimo agrega automáticamente la transitividad utilizando el algoritmo de Warshall. Sin embargo especificar un orden válido sigue siendo complicado. El algoritmo permite generar automáticamente una serie de órdenes que satisfacen todas las propiedades necesarias.

Los órdenes propuestos son:

- Orden por niveles

Llamamos *nivel* de una fórmula f_i a la cantidad de valuaciones en que f_i es verdadera. Si asociamos el valor verdadero al número 1 y el valor falso al número 0, el $nivel(fi)$ sería la cantidad de unos de la tabla de verdad de f_i.

En el ordenamiento por niveles propuesto si $nivel(fi) < nivel(fj)$ entonces $fi \prec fj$. El algortimo empleado crea explícitamente un par $fi \prec fj$ si $nivel(fi)+1 = nivel(fj)$ y luego se aplica transitividad completando los pares en la relación $\prec$. No se relacionan las fórmulas que tienen igual nivel.

Para ver que este orden satisface *continuing up* hace falta ver que para todo $x, y, z \in A$, si $x \prec y$ y $y \vdash z$, entonces $x \prec z$.

Dado que $x < y$, por la definición del orden tenemos que el $nivel(x) < nivel(y)$. Como $y \vdash z$, y cubre todos los ceros de z, por lo que la cantidad de unos de y es menor o igual que la cantidad de unos de z, es decir, $nivel(y) \preceq nivel(z)$.

Si $nivel(y) = nivel(z)$ entonces $y = z$, dado que la única forma en que una fórmula implique a otra que tenga la misma cantidad de ceros es que sea la misma, por lo tanto se cumple que $x \prec z$.

Si $nivel(y) < nivel(z)$ entonces, por la forma en que se contruye el orden, resulta que $y \prec z$. Por lo tanto si $x \prec y$ y $y \prec z$, entonces $x \prec z$ por transitividad, con lo cual se cumple la propiedad de *continuing up*.

Para verificar *virtually connected* tenemos que verificar que para todo $x, y, z \in A$, si $x \prec y$ entonces $x \prec z$ o $z \prec y$.

Analicemos las distintas posibilidades:

Si $nivel(z) < nivel(x)$ entonces, por definición del orden, $z \prec x$ y por hipótesis, $x \prec y$. Usando la transitividad tenemos que $z \prec y$, con lo cual se cumple la propiedad.

Si $nivel(z) = nivel(x)$ entonces, como $x \prec y$, $nivel(z) < nivel(y)$ por definición del orden, de donde se deduce que $z \prec y$, es decir que también cumple la propiedad.

Por último, si $nivel(x) < nivel(z)$, por definición del orden, resulta que $x \prec z$, que era lo que queríamos probar. Por lo tanto la propiedad se verifica en todos los casos.

- Orden Total

En este ordenamiento se agregan las mismas relaciones que el ordenamiento anterior pero además se relacionan las fórmulas que tienen el mismo nivel, es decir, si $nivel(f_i) < nivel(f_j)$ entonces $f_i \prec f_j$ y si $nivel(f_i) = nivel(f_j)$ entoces $f_i \prec f_j$ ó $f_j \prec f_i$

Podemos usar un "orden total" porque trabajamos con clases de equivalencia, en caso contrario no sería posible dado que no se cumpliría la propiedad *continuing up*, que no permite relacionar fórmulas lógicamente equivalentes.

La demostración de *continuing up* para este orden es exáctamente igual a la del orden anterior.

En cuanto a la propiedad de *virtually connected* tenemos que verificar que para todo $x, y, z \in A$, si $x \prec y$ entonces $x \prec z$ o $z \prec y$.

Analicemos otra vez las distintas posibilidades:

Si $nivel(z) < nivel(x)$ entonces, por definición del orden, $z \prec x$ y por hipótesis, $x \prec y$. Usando la transitividad tenemos que $z \prec y$, con lo cual se cumple la propiedad.

Si $nivel(z) = nivel(x)$ entonces, por definición del orden, $z \prec x$ ó $x \prec z$. Si $x \prec z$ se cumple la propiedad, con lo cual no hay nada que demostrar. Por otra parte si $z \prec x$, y como también sabemos que $x \prec y$ por hipótesis, por transitividad resulta que $z \prec y$, por lo tanto también cumple la propiedad.

Por último, si $nivel(x) < nivel(z)$, por definición del orden, resulta que $x \prec z$, que era lo que queríamos probar. Por lo tanto la propiedad se verifica en todos los casos.

5.2 Ejemplo de un elemento "unsafe" que reaparece en el resultado.

En este ejemplo se muestra cómo un elemento que no es safe puede peretencer a $K - x$.

Trabajando con dos variables proposicionales, y con la base $\{a, b\}$ y contrayendo por la fórmula $a \wedge b$.

Eligiendo un orden total como el que sigue:

$a \wedge b \prec (a \longleftrightarrow b) \prec a \prec b \prec (a \vee -b) \prec (-a \vee b) \prec (a \vee b) \prec \top$

Se obtienen las siguientes demostraciones minimales.

1- $\{a \wedge b\}$ mínimo:$a \wedge b$

2- $\{(a \longleftrightarrow b), a\}$ mínimo: $(a \longleftrightarrow b)$

3- $\{(a \longleftrightarrow b), b\}$ mínimo: $(a \longleftrightarrow b)$

4- $\{(a \longleftrightarrow b), a|b\}$ mínimo:$(a \longleftrightarrow b)$

5- $\{a, b\}$ mínimo: a

6- $\{a, -a \vee b\}$ mínimo:a

7- $\{b, a \vee -b\}$ mínimo: b

8- $\{a \vee -b, -a \vee b, a \vee b\}$ mínimo: $a \vee -b$

Se detectan los elementos safe: $\{(-a \vee b), (a \vee b), \top\}$

Dado que $\{(-a \vee b), (a \vee b)\} \vdash b$, b reaparece y $b \in Cn(K/(a \wedge b))$

Si en el orden elegido hubiera resultado $(a \vee b)$ el menor entre $\{(a \vee -b), (-a \vee b), (a \vee b)\}$

Entonces la fórmula que reaparecería sería: $(a \rightarrow b) \wedge (b \rightarrow a)$

Si en el orden elegido hubiera resultado $(-a \vee b)$ el menor entre $\{(a \vee -b), (-a \vee b), (a \vee b)\}$

Entonces la fórmula que reaparecería sería: a

Como ya se ha mencionado, cuando el orden es vacío tenemos que $K - x = K/x$ y no es necesario calcular $Cn(K/x)$

6 Referencias

References

[1] Carlos Alchourrón and David Makinson, On the logic of theory change: Safe Contraction. *Studia Logica,* 44:405-422, 1985

[2] Carlos Alchourrón, Peter Gärdenfors and David Makinson. On the logic of theory change: Partial Meet Contraction and Revision Functions. *Journal of Symbolic Logic,* 50:510-530, 1985.

[3] Carlos Alchourrón and David Makinson, Maps between some differents kinds of contraction functions: The finite case. *Studia Logica,* 45:187-198, 1986

[4] David Makinson. On the Status of the postulate of recovery in the logic of theory change. *Journal of Philosophical Logic,* 16:383-394, 1987

[5] Peter Gärdenfors. *Knowledge in Flux: Modeling the Dynamics States.* Bradford Books/MIT Press, Cmabridge (Mass.), 1988.

[6] Michael R. Garey, David S. Johnson.*Computers and Intractability. A Guide to the Theory of NP-Completeness.*W.H. Freeman.1979

[7] Sven Ove Hansson, *Revision of Belief Sets and Belief Bases*

[8] Hans Rott. On the logic of theory change: More maps between different kinds of contraction functions. In Peter Gärdenfors, editor, *Belief Revision,* number 29 in Cambridge Tracts in Theoretical Computer Science, pages 122-141. Cambridge University Press, 1992

[9] Ravi Sethi Alfred V. Aho and Jeffrey Ullman. *Compilers. Principles, Techniques and Tools.* Addison Wesley Publishing Company, 1986.

[10] Mary Anne Williams. *SATEN: Information Extraction and Revision Engine.* Http://ebusiness.newcastle.edu.au/systems/saten.html

[11] Christian Durr, *Construcción de Funciones Analíticas para el Cambio de Teorías,* Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina, 2001

7 Agradecimientos

A Carlos Areces, por la recomendación de la estrategia principal del algoritmo.

A Christian Durr, por facilitarnos sus algoritmos, sus programas fuentes y la explicación de los mismos.

A Vero Becher por su constante apoyo y dedicación.