Implementación de Operadores de Contracción Múltiple para Bases de Creencias Horn

Néstor Jorge Valdez† and Marcelo A. Falappa‡

† Dpto. de Ciencias de la Computación, Fac. de Ciencias Exactas y Naturales
Universidad Nacional de Catamarca (UNCa)
Av. Belgrano 300 - San Fernando del Valle de Catamarca
Tel.: (03834)420900 / Cel: (03834) 154591186
e-mail: njvaldez@c.exactas.unca.edu.ar
‡ Laboratorio de Investigación y Desarrollo en Inteligencia Artificial
Dpto. de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur,
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
Av. Alem 1253, (B8000CPB) Bahía Blanca, Argentina
Tel: (0291)4595135 / Fax: (0291)4595136
e-mail: mfalappa@cs.uns.edu.ar

Resumen En la teoría de cambio de creencias, uno de los problemas claves es la implementación computacional en cuanto a operaciones de contracción múltiple en bases de creencias. Algunas aproximaciones pueden basarse en construcciones de tipo kernel o en construcciones de tipo partial meet. En este paper, pretendemos resolver este problema mediante el estudio de algoritmos para aplicar estas construcciones en cláusulas Horn. Las semejanzas observadas entre estos algoritmos sugiere que están sensiblemente relacionados. Mostramos su funcionalidad con ejemplos para el cálculo de los conjuntos de restos (remainder set) y los conjuntos kernel (kernel set) con cláusulas Horn. La ventaja de elegir cláusulas de Horn es que su estructura simple y propiedades únicas se pueden utilizar para mejorar la eficiencia del algoritmo.

Keywords: Cambio de Creencias, Operadores de Contracción Múltiple, Cláusulas Horn.

1. Introducción

1.1. Motivación

Para implementar las operaciones de cambio, normalmente es necesario representar el conocimiento mediante bases de creencias puesto que si contamos con un lenguaje (al menos proposicional) lo suficientemente declarativo, no es posible tratar con conjuntos clausurados. Sabemos que el modelo AGM asume una lógica subyacente que es al menos tan expresiva como la lógica proposicional. Debido a este supuesto, esta teoría no puede ser aplicada a los sistemas con

lógicas subyacentes declarativamente más ricas que la lógica proposicional (e.g., los sistemas de Inteligencia Artificial). Como muchos sistemas operan bajo lógicas no clásicas, el marco AGM no puede ser directamente aplicado a los mismo y, por ello, se impone la necesidad de estudiar la adaptación del marco AGM a estas lógicas no clásicas. Diversas investigaciones estudiaron las operaciones de cambio introducidas en AGM, pero no necesariamente bajo lógica proposicional, y entre ellas se destacan las lógicas Horn.

El problema de calcular todos los conjuntos ya sea maximales o minimales que no impliquen o impliquen respectivamente a un conjunto de fórmulas arbitrarias es muy complejo. En este trabajo, elegimos cláusulas de Horn, un subconjunto especial de lenguaje de primer orden como objetivo de la investigación. Una cláusula de Horn es una sentencia del lenguaje de primer orden de la forma: $\neg a_1 \vee ... \vee \neg a_n \vee b$ o $\neg a_1 \vee ... \vee \neg a_n$, donde $a_1, ..., a_n, b$ son sentencias atómicas. La ventaja de elegir cláusulas de Horn es que su estructura simple y propiedades únicas se pueden utilizar para mejorar la eficiencia del algoritmo. Y puesto que la cláusula Horn es la base del lenguaje PROLOG, las bases de conocimientos escrito en PROLOG puede ser visto como un conjunto de cláusulas de Horn, por lo que es posible utilizar un algoritmo para calcular una contracción en la base y mantener la consistencia en la misma.

1.2. Extensión de Contracción de Creencias Horn de Una Sentencia a Múltiples Sentencias

Por cuestiones de espacio en este paper, obviaremos aspectos teóricos relacionados a operaciones de contracción de creencias Horn de una sentencias para enfocarnos en las contracciones de sentencias múltiples. Para ampliar su lectura sugerimos considerar la publicación [7].

En cuanto a una representación de resultados para el tipo de contracción múltiple denominada *package contraction* bajo lógicas Horn, existen diversas maneras en que la contracción Horn *(simple y múltiple)* se diferencia de la contracción AGM. Una de ellas es la ausencia del postulado de *recovery*. Esto podría deberse a que, al menos para cláusulas Horn, el postulado *recovery* resulta inapropiado. Otra manera, es que en la contracción Horn, el resultado de la misma puede ser definido mediante un conjunto de restos. Otra diferencia es que con cláusulas Horn uno tiene dos clases de operadores de contracción: *entailment-based contraction* (*e-contraction*) e *inconsistency-based contraction* (*i-contraction*).

1.3. Preliminares

Consideramos un lenguaje proposicional $\mathcal{L}$, sobre un conjunto de literales $\mathbf{P} = \{p, q, \ldots\}$, con semánticas de un modelo teórico estándar. Los caracteres griegos en minúsculas $\varphi, \psi, \ldots$ denotan fórmulas y los caracteres en mayúsculas $X, Y, \ldots$ denotan conjuntos de fórmulas. Una cláusula Horn es una cláusula con a lo sumo un literal positivo. Una fórmula Horn es una conjunción de cláusulas Horn. Una teoría Horn es un conjunto de fórmulas Horn. El lenguaje Horn

$\mathcal{L}_H$ es una restricción de $\mathcal{L}$ para fórmulas Horn. La lógica Horn obtenida de $\mathcal{L}_H$ tiene la misma semántica que la lógica proposicional obtenida de $\mathcal{L}$, pero restringida para fórmulas Horn y sus derivables. La consecuencia lógica clásica y su equivalencia lógica se denota por $\vdash$ y $\equiv$ respectivamente. Cn es el operador de consecuencia tal que $Cn(X) = \{\varphi \mid X \vdash \varphi\}$. La consecuencia lógica bajo lógica Horn se denota por $\vdash_H$ y así el operador de consecuencia Cn_H bajo lógica Horn es tal que $Cn_H(X) = \{\varphi \mid X \vdash_H \varphi\}$. Los conjuntos clausurados se representarán mediante letras en negrita. Por ejemplo, si $\mathbf{K}$ es un conjunto de creencias entonces $\mathbf{K} = Cn(\mathbf{K})$.

Este paper está organizado de la siguiente manera: en la Sección 2 se presentan dos modelos básicos de construcción de contracciones múltiples, los que han motivado esta investigación. En las secciones 3 y 4 presentamos lo que constituye las contribuciones de este artículo: un algoritmo y su heurística para encontrar un elemento del kernel; la definición de un corte minimal y presentamos el algoritmo de Reiter; y la presentación de los algoritmos para encontrar los conjuntos kernel y de restos respectivamente. Estos algoritmos están claramente relacionados uno con otro. Finalmente, en la última sección presentamos las conclusiones y trabajos futuros.

2. Modelos de Construcción de Contracción Múltiple

2.1. Partial Meet Multiple Contraction

Teniendo en mente los conceptos básicos de funciones partial meet contraction referido a una única sentencia [1], presentaremos los conceptos fundamentales de las *partial meet multiple contractions*.

Definición 1 *[5] [4] Dado un conjunto de creencias* $\mathbf{K}$ *y un conjunto de sentencias* B*, según la definición indicada anteriormente, los restos* remainders *de* $\mathbf{K}$ *por* B *son los subconjuntos maximales de* $\mathbf{K}$ *que no implican algún elemento de* B*. Además, el conjunto formado por todos los restos* remainders *(de* $\mathbf{K}$ *por* B*) es el conjunto de restos de* $\mathbf{K}$ *por* B *y es denotado por* $\mathbf{K} \perp B$*.*

Definición 2 *[5] [4] Sea* $\mathbf{K}$ *un conjunto de creencia,* B *un conjunto de sentencias y* $\mathbf{K} \perp B$ *el conjunto de restos de* K *con respecto a* B*. Una* package selection function *para* $\mathbf{K}$ *es una función* γ *tal que para todos los conjuntos de sentencias* B*:*

1. *Si* $\mathbf{K} \perp B$ *es no-vacío, entonces* $\gamma(\mathbf{K} \perp B)$ *es un subconjunto no vacío de* $\mathbf{K} \perp B$*, y*
2. *Si* $\mathbf{K} \perp B$ *es vacío, entonces* $\gamma(\mathbf{K} \perp B) = \{\mathbf{K}\}$*.*

Entonces, la definición de *partial meet multiple contraction* producto de la generalización de *partial meet contraction* para el caso de contracciones por conjuntos de sentencias es:

Definición 3 ***(Partial meet multiple contraction** [5] [4] Sea* $\mathbf{K}$ *un conjunto de sentencias y sea* γ *una package selection function para* $\mathbf{K}$*. La partial meet multiple contraction de* $\mathbf{K}$ *generada por* γ *es la operación* $\div_\gamma$ *tal que para cualquier conjunto de sentencias de* B*:*

$$\mathbf{K} \div_\gamma B = \bigcap \gamma(\mathbf{K} \perp B)$$

Una multiple contraction function $\div$ *de* $\mathbf{K}$ *es una partial meet multiple contraction si y solamente si existe alguna package selection function* γ *tal que* $\mathbf{K} \div B = \mathbf{K} \div_\gamma B$ *para cualquier conjunto de sentencias* B.

Las definiciones de los dos casos límites particulares de *partial meet contractions* son:

Definición 4 *Sea* $\mathbf{K}$ *un conjunto de creencias.*

1. *Una multiple contraction function* $\div$ *en* $\mathbf{K}$ *es una maxichoice multiple contraction si y solamente si es una partial meet multiple contraction generado por un package selection function* γ *tal que para todos los conjuntos* B*, el conjunto* $\gamma(\mathbf{K} \perp B)$ *tiene exactamente un elemento.*
2. *La full meet multiple contraction en* $\mathbf{K}$ *es el partial meet multiple contraction* $\doteqdot$ *que es generado por la package selection function* γ *tal que para todos los conjuntos* B*, si* $\mathbf{K} \perp B$ *es no-vacío, entonces* $\gamma(\mathbf{K} \perp B) = \mathbf{K} \perp B$*, i.e., la multiple full meet contraction* $\doteqdot$ *es la operación de contracción en* $\mathbf{K}$ *definido por:*

$$\mathbf{K} \doteqdot B = \begin{cases} \bigcap \mathbf{K} \perp B & \text{si } B \bigcap Cn(\varnothing) = \varnothing \\ \mathbf{K} & \text{en caso contrario} \end{cases}$$

para cualquier conjunto B.

2.2. Kernel Multiple Contraction

Se presenta a continuación la definición de *contracción múltiple kernel* que resulta ser una generalización de la operación de contracción kernel para una sola sentencia, pero que se refiere para contracciones por conjuntos de sentencias.

Definición 5 [3] *Sea* A *y* B *dos conjuntos de sentencias. El package kernel set de* A *con respecto a* B*, denotado* $A \Subset B$ *es el conjunto tal que* $X \in A \Subset B$ *si y solamente si:*

1. $X \subseteq A$.
2. $B \cap Cn(X) \neq \varnothing$.
3. *Si* $Y \subset X$*, entonces* $B \cap Cn(Y) = \varnothing$.

Esta definición es más general, pues A no necesariamente es un conjunto de creencias. La definición de package incision function para un conjunto A, que resulta en una función que selecciona al menos un elemento de cada uno de los conjuntos en $A \Subset B$, para cualquier conjunto B.

Definición 6 [3] *Una función* σ *es una función de incisión para* A *si y solamente si, para cualquier* B*:*

1. $\sigma(A \Subset B) \subseteq \bigcup A \Subset B$.
2. *Si* $\varnothing \neq X \in A \Subset B$*, entonces* $X \cap \sigma(A \Subset B) \neq \varnothing$.

Definición 7 *(**Kernel Multiple Contraction** [**3**]) Sea* σ *una incision function para* A*. La kernel multiple contraction* $\approx_\sigma$ *para* A *basado en* σ *está definida como sigue:*

$$A \approx_\sigma B = A \backslash \sigma(A \Subset B).$$

Una multiple contraction function $\div$ *para* A *es una kernel multiple contraction si y solamente si existe alguna package incision fuction* σ *para* A *tal que* $A \div B = A \approx_\sigma B$ *para cualquier* B.

3. Perspectiva Computacional para Contracciones Múltiples de Creencias Mediante Conjuntos Kernel

En la teoría de cambio de creencias, el cálculo de los conjuntos de restos (*remainder sets*) y los conjuntos kernel (*kernel sets*) para las operaciones de contracción (con respecto a una o múltiples sentencias como entrada) resulta ser un problema clave. En esta investigación, tratamos este problema mediante el estudio de algoritmos que calculan las *contracciones múltiples* para cláusulas Horn de una base de creencias. Estos algoritmos son adaptaciones de algoritmos bien conocidos en el campo de la Inteligencia Artificial.

Los algoritmos para encontrar los conjuntos de restos y kernel están muy relacionados. Es posible demostrar que dado uno de ellos, es posible derivar el otro algoritmo sin más debido a su inferencia teórica. Esta relación entre conjuntos de restos y kernel fue ampliamente estudiada por Falappa [2]. Además de esta estrecha relación, el conjunto de restos normalmente depende exponencialmente de la cantidad de elementos de la base de creencias, mientras que el conjunto de kernels depende polinomialmente de la cantidad de elementos de la base de creencias.

En los trabajos que anteceden a esta investigación [7], se consideró el cambio de creencias bajo una lógica más restringida que la lógica proposicional clásica del marco AGM como es la lógica Horn. Tuvimos que examinar diversos resultados existente en el área, todos ellos referente al estudio de cambio de creencias bajo lógica Horn de las operaciones de contracción con respecto de una sola sentencia. De esta manera, pudimos generalizar algunos modelos para los casos de contracción múltiple bajo lógica Horn. Partiendo de esta base, nuestro primer paso consistirá en plantear metódicamente el procedimiento computacional para una operación múltiple de tipo *kernel*.

3.1. Cálculo de los Φ-kernels

Para obtener los Φ-kernel recurrimos a un lenguaje de programación en lógica como PROLOG. Una de las grandes ventajas que proporciona PROLOG es la facilidad de definir *metaintérpretes*. Los metaintérpretes de PROLOG son intérpretes de PROLOG escritos en lenguaje PROLOG. A continuación, se presenta *parte del código* en PROLOG del sistema desarrollado donde se observa el metaintérprete que obtiene los árboles de prueba de la consulta Φ (un conjunto de cláusulas Horn de entrada). La generalización de este metaintérprete mediante el predicado de segundo orden del lenguaje PROLOG (setof) permite obtener todas las pruebas de la consulta Φ.

Metaintérprete Obtener-Kernel
Hallar un subconj. A **de** K **tal que** $A \vdash \Phi$
Sea $\phi \in \Phi$

obtener-kernel-bd :- empezar, nl, tab(5), write('Ingrese $\phi \in \Phi$ kb1(?)): '),
read(J), nl,

```
setof(L,solve(J,L),CON), nl, nl, write('- Los Conjuntos Kernels φ son: '), nl, nl,
lista1(CON).

empezar :- write('Base de Conocimientos: '), read(BC), consult(BC), !,
nl, nl.
empezar :- nl, tab(10), write('ERROR: Base de Conocimientos no encontrada'), nl.

builtin(A is B):- write(A), write(B).
/*builtin(A > B).
builtin(read(X)).
builtin(write(X)).
builtin(integer(X)).
builtin(functor(T,F,N)).*/
builtin(clause(A,B)):- write(A), write(B).
builtin(builtin(X)):- write(X).

solve(true,[ ]):- !.
solve((A,B),L):- !, solve(A,LA), solve(B,LB), conc(LA,LB,L).
solve(A, (A:-builtin)):- builtin(A) , !, A.
solve(A,[A:-B | L]):- clause(A, B), solve(B,L).

conc([X | Xs],Ys,[X | Zs]):- conc(Xs,Ys,Zs).
conc([ ],X,X).

lista1([ ]).
lista1([H | T]):- tab(5), write(H), lista1(T).
```

Probar que $A \vdash \Phi$ es equivalente a demostrar que $A \cup \{\neg\Phi\} \vdash \perp$, siempre que la derivación no genere ramas infinitas.

Por último, para encontrar el conjunto de los conjuntos kernel $K \Vdash \Phi$ el sistema ejecuta una iteración del tipo **for** $\phi \in \Phi$, convocando a la función **obtener-kernel-bd** y almacenando al final de cada iteración la **lista1** en una **listaGeneral** que contendrá todos los conjuntos kernel obtenidos en cada iteración a través de **lista1**.

Observación: la lista **listaGeneral** estará compuesta de las sublistas **lista1** (obtenidas en cada iteración).

3.2. Los cortes minimales

El metaintérprete *Obtener-Kernel* es usado para obtener todos los kernel. Ahora, para determinar los cortes minimales podemos usar una idea presentada por Wassermann en [8] que está basado en el algoritmo de Reiter [6]. El *algoritmo Reiter* encuentra todos los cortes minimales de un conjunto.

Definición 8 Corte Minimal *Un corte para un conjunto de conjuntos $\mathcal{K}$ es un conjunto K que intersecta cada conjunto en el conjunto $\mathcal{K}$. Un corte C es mi-*

nimal si y solo si no hay un corte C' adecuadamente contenida en C i.e., si $C' \subset C$, entonces C' no es un corte para $\mathcal{K}$.

3.3. Cálculo de los Cortes Minimales

Por cuestión de espacio en el presente trabajo, solamente presentaremos el algoritmo (procedimiental) la que resulta una versión simplificada del algoritmo mencionado.

El algoritmo empieza extrayendo un elemento de **listaGeneral** que contiene todos los subconjuntos kernel obtenidos mediante el metaintérprete **Obtener-Kernel** presentado en la sección anterior. En cada iteración obtenemos un corte almacenado en la variable **Cort**.

Algoritmo que calcula los cortes minimales

1. Encontrar los cortes $K \Vdash \Phi$
2. **for** elemento1 (una sublista) $\in$ **listaGeneral**
3. **do** $Cort \leftarrow \varnothing$
4. $pila \leftarrow pila$ vacía
5. **for** elemento $\in$ elemento1 (una sublista)
6. $S \leftarrow$ elemento
7. **for** $s \in S$
8. **do** insertar $\{s\}$ en la parte superior de la $pila$
9. **while** $pila$ no esté vacía
10. **do** $Vn \leftarrow$ último elemento de la $pila$
11. eliminar último elemento de la $pila$
12. **if** $\exists C \in Cort$ tal que $C \subseteq Vn$
13. **then** continuar
14. **else if** $\exists S \in elemento1$ tal que $Vn \cap S = \varnothing$
15. **then for** $s \in S$
16. **do** insertar $Vn \cup \{s\}$ en la parte superior de la $pila$
17. **else**
18. $Cort \leftarrow Cort \cup \{Vn\}$
19. **return** $Cort$

La variable Vn es un corte minimal del kernel y recibe un elemento de la pila. Si hay algún corte que contiene Vn, entonces Vn no es minimal y Vn es descartado. De lo contrario, el algoritmo controla si hay algún elemento de la sublista *elemento1* que no intersecta con Vn. Si no hay ninguno, entonces Vn es un corte minimal y es agregado al conjunto $Cort$ de cortes minimales. De lo contrario, por cada elemento s del conjunto que no intersecta con Vn tenemos que $Vn \cup \{s\}$ es un potencial corte minimal de *elemento1*. Cada $Vn \cup \{s\}$ es agregado a la pila a fin de ser verificado en la siguiente iteración. Al final de este proceso, $Cort$ contiene todos los cortes minimales de **listaGeneral**.

Presentamos un algoritmo que ha sido bien estudiado y aplicado en lenguaje de programación lógica (PROLOG). No vamos a entrar en los detalles de este algoritmo aquí. Sin embargo, dado que la consistencia es indecidible en cláusulas de Horn, proponemos un algoritmo interactivo para calcular los subconjuntos

minimales del kernel. El algoritmo registra a todos los candidatos para formar parte del conjunto kernel, así también, queda abierta la posibilidad a que el usuario tome la decisión de descartar a algunos.

El algoritmo presentado aquí cumple las siguientes propiedades:

- la obtención de los cortes minimales permiten garantizar la consistencia del conjunto de cláusulas Horn (K). Es decir, el algoritmo realiza el control usando el método de SLD-refutación en el conjunto de cláusulas Horn,
- el algoritmo define un mecanismo de selección que determina que sentencias (ϕ-kernel) considerar y cuales no, así también, su criterio de preferencia (sintáctico),
- Una de las características de este proceso, es que trabaja directamente sobre el motor de inferencia de PROLOG. Este método selecciona la submeta de más a la izquierda y utiliza como regla de búsqueda la primer cláusula cuya cabeza unifica con la submeta a resolver (resolución-SLD).
- Estos algoritmos definen la operación del cálculo del conjunto kernel sobre una base de creencias a partir de sentencias simples. Es decir, las sentencias que se consideran serán restringidas a átomos sin variables libres y a conjunciones de éstos,
- Debido al lenguaje considerado y sus mecanismos de uso (SLD-Refutación o SLD-Resolución), no es posible generar información falsa. Es decir, que tendremos siempre una BC consistente y, los procedimientos posibles de una BC serán únicamente a partir de literales positivos (átomos).

4. Perspectiva Computacional para Contracciones Múltiples de Creencias Mediante Conjuntos de Restos

Para encontrar el conjunto de restos, necesitamos *invertir* el procedimiento presentado en la sección anterior. Para encontrar un elemento de resto (*remainder*) sumamos elementos de K a un conjunto A verificamos si Φ no es todavía consecuencia. La siguiente definición será útil en nuestro procedimiento.

Definición 9 *Un conjunto Φ es una tautología si y solo si para $\forall \psi \in \Psi$ decimos que, para cada valuación ψ hay un $\varphi \in \Phi$, tal que $\psi \vdash \varphi$.*

Se puede verificar que $K \perp \Phi = \varnothing$ si y solo si Φ es una tautología.

Aquí también, recurrimos a un metaintérprete **Obtener-Restos** de PROLOG para hallar los conjuntos de restos.

Probar que $A \nvdash \Phi$ es equivalente a demostrar que $A \cup \{\Phi\} \vdash \top$, siempre que la derivación no genere ramas infinitas. Por último, el procedimiento también puede generalizarse mediante un mecanismo de negación por falla de PROLOG, obteniento TODOS los restos.

Para encontrar los otros elementos del conjunto de restos *remainder set* usaremos una adaptación del *algoritmo de Reiter* (considerado para el cálculo del Kernel en la sección anterior). En este caso, el procedimiento para encontrar el conjunto de restos es muy similar al procedimiento del algoritmo para encontrar

el kernel. Vn es ahora un posible corte en el conjunto $\{K\backslash Y : Y \in K\bot\Phi\}$. En cada iteración si Vn no es un corte, entonces el algoritmo encuentra los elementos de $K\bot\Phi$ que incluyen Vn.

4.1. Cálculo de los Cortes Maximales

Resulta evidente la similitud de sus procedimientos. Podemos decir, que intuitivamente cada uno de estos procedimientos es la inversa del otro. La existencia de una relación formal entre ellos fue estudiado por Falappa [2]. Sin embargo, en ese trabajo, ellos centran su atención en la relación entre la función de incisión y la función de selección. Por último, aquí nos referimos a la relación formal entre conjunto de restos *remainder set* y kernel solamente.

Algoritmo que calcula los cortes maximales
1. Encontrar los cortes de $K\bot\Phi$
2. **for** elemento1 (una sublista) $\in$ **listaGeneral**
3. **do** $Cort \leftarrow \varnothing$
4. $pila \leftarrow pila$ vacía
5. **for** elemento $\in$ elemento1 (una sublista)
6. $S \leftarrow$ elemento
7. **for** $s \in K\backslash S$
8. **do** insertar $\{s\}$ en la parte superior de la $pila$
9. **while** $pila$ no esté vacía
10. **do** $Vn \leftarrow$ último elemento de la $pila$
11. eliminar último elemento de la $pila$
12. **if** $\exists C \in Cort$ tal que $C \subseteq Vn$
13. **then** continuar
14. **else if** $\exists S \in elemento1$ tal que $Vn \cap S = \varnothing$
15. **then for** $s \in K\backslash S$
16. **do** insertar $Vn \cup \{s\}$ en la parte superior de la $pila$
17. **else**
18. $Cort \leftarrow Cort \cup \{Vn\}$
19. **return** $Cort$

Más allá de la intuitiva relación entre las contracciones kernel y partial meet, en ciertos casos es mucho más rápido calcular el kernel, como así también, hay casos en que es más rápido el cálculo del conjunto de restos con respecto al cálculo del kernel.

5. Conclusión y Trabajos Futuros

Traducir al terreno computacional los procedimientos de la construcción de la revisión y la contracción múltiple de una base donde se determina el kernel y el conjunto de restos, es muy complejo y difícil. En este paper, hemos propuesto algoritmos (porque no interactivos!) para encontrar tanto el kernel como el conjunto de restos de una base de la creencia como entrada. Estos algoritmos son

adaptaciones de algoritmos usados en otros campos de la inteligencia artificial, tales como, la depuración ontológica y el diagnóstico [6]. Para profundizar más su lectura acerca de estos algoritmos y su aplicación en la contracción múltiple de base de la creencia se sugiere [8].

Los algoritmos hacen uso de todo el potencial de la programación lógica (el uso de la lógica matemática para la programación). Esto hace que nuestros algoritmos sean fáciles de integrar con PROLOG y de esta manera proporcionar la capacidad de mantener la consistencia de la base de conocimientos escrito en PROLOG. Sin embargo, hay que señalar que los algoritmos tiene sus limitaciones. Debido a la indecisión de la consistencia, por lo que es posible que se requiera la intervención del usuario en caso de que las cláusulas de Horn contengan variables.

También, mostramos como los algoritmos para encontrar el kernel y el conjunto de restos están estrechamente relacionados. Es decir, que dado uno el otro puede ser derivado de él. La relación entre la kernel contraction y la partial meet contraction fue primeramente estudiado en [2].

Para futuros trabajos, vamos a considerar el problema de calcular las restantes operaciones de contracción múltiples con cláusulas Horn. Otro problema interesante, sería determinar en que subconjuntos de un lenguaje de primer orden podemos calcular las contracciones múltiples de forma totalmente automática, y agregado a esto, estudiar la complejidad computacional del problema en estos casos.

Referencias

1. Alchourrón, Gärdenfors, and Makinson. On the logic of theory change: Partial meet contraction and revision functions. *The Journal of Symbolic Logic*, 50, 1985.
2. Falappa, Fermé, and Kern-Isberner. On the logic of theory change: Relations between incision and selection functions. *In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*, pages 402–406, 2006.
3. Fermé, Saez, and Sanz. Multiple kernel contraction. *Studia Logica*, 73:183–195, 2003.
4. Fuhrmann and Hansson. A survey of multiple contractions. *Journal of Logic, Language and Information*, pages 39–74, 1994.
5. Hansson. New operators for theory change. *Theoria*, 55:114–132, 1989.
6. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32.
7. Valdez and Falappa. Dinámica de conocimiento: Contracción múltiple en lenguajes horn. *XIX Congreso Argentino de Ciencias de la Computación, XIV Workshop Agentes y Sistemas Inteligentes (WASI), CACiC'2013*, 2013.
8. Wassermann. An algorithm for belief revision. *In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning. Breckenridge, Colorado, USA, Abril, 15-20 2000. Morgan Kaufmann*, pages 345–352, 2000.